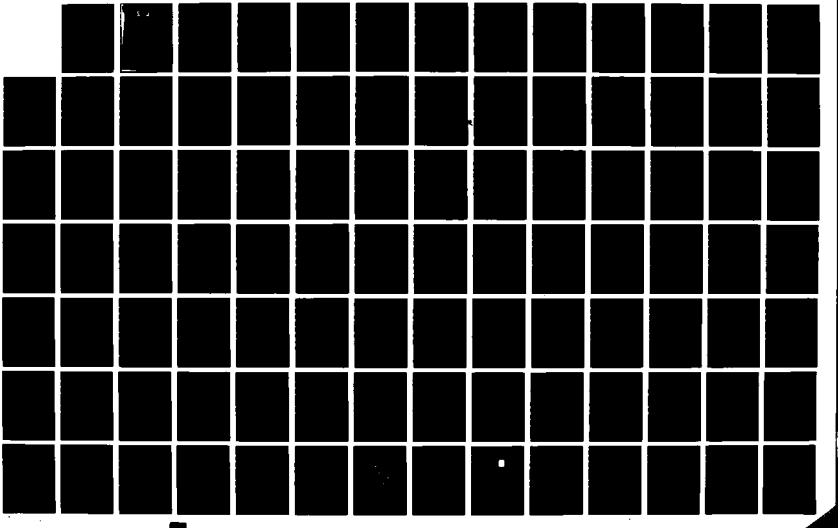
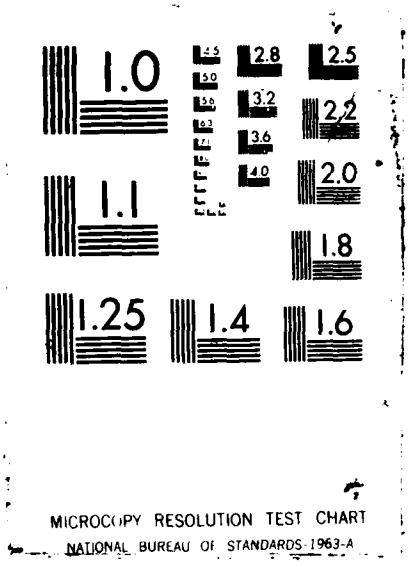
AD-A185 571 LOGIC PROGRAMMING AND KNOWLEDGE MAINTENANCE(U) SYRACUSE 1/2
UNIV NY SCHOOL OF COMPUTER AND INFORMATION SCIENCE
K A BOWEN 13 AUG 87 AFOSR-TR-87-1304 #AFOSR-82-0292
UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNLCLASSIFIED

AD-A185 571

DTIC FILE COPY

(2)

REPORT DOCUMENTATION PAGE

DTIC ELECTE OCT 0 1 1987 S D

1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 87-1304
6a NAME OF PERFORMING ORGANIZATION: SYRACUSE UNIVERSITY	6b OFFICE SYMBOL (If applicable):
6c ADDRESS (City, State, and ZIP Code): SCHOOL IF COMPUTER AND INFORMATION SCIENCES 313 LINK HALL, SYRACUSE UNIVERSITY SYRACUSE, NY 13210	7a NAME OF MONITORING ORGANIZATION: AFOSR/NM 7b ADDRESS (City, State, and ZIP Code): BLDG 410 BOLLING AFB, DC 20332-6448
8a NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR	8b OFFICE SYMBOL (If applicable): NM
8c ADDRESS (City, State, and ZIP Code): BLDG 410 BOLLING AFB, DC 20332-6448	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR_82-0292
	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO 61103F; PROJECT NO 2304; TASK NO A7; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification): LOGIC PROGRAMMING AND KNOWLEDGE MAINTENANCE

12 PERSONAL AUTHOR(S): PROFESSOR KENNETH A BOWEN

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
FINAL	FROM 84/10/30 TO 86/11/30	1987 Aug 13	151

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The focus of this work was to study large volatile knowledge bases. The research involved developing extensions to logic programming systems in the form of a metalanguage, by studying to what extent frames and semantic nets could be employed. The management of consistency and integrety under change using a maetalanguage was analyzed. This research produced a rule-based deductive programming lanaguage, called metaProglog, which enhances Prolog's ability to manipulate the databases themselves and to reason about them. This was accomplished by regarding databases (or theories) as first-class objects capable of being passed as arguments. Four papers were published under this grant, including "Meta-kavek programming and knowledge representation" and "metaProlog: A metalevel extension to Prolog".

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL
James M Crowley Maj	767-5025	NM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

AFOSR-TR- 87-1304

LOGIC PROGRAMMING AND KNOWLEDGE BASE MAINTENANCE

Final Report
to the
Air Force Office of Scientific Research
Grant AFOSR-82-0292

Principal Investigator:

Professor Kenneth A. Bowen
School of Computer and Information Science
313 Link Hall, Syracuse University,
Syracuse, NY 13210
(315)-423-3564

Contents

A.	Introductory Material and Achievements	2
	Abstract and Summary	2
	1. Logic Programming	4
	2. Logic and Knowledge Bases	12
	3. Project Plan & Achievements	17
	4. Students & Publications	22
B.	metaProlog: Design & Application	23
	1. Introduction	23
	2. Meta-level Programming and metaProlog	28
	3. The metaProlog System	35
	4. Quantification and Naming: Language Foundations	43
	5. Programming Examples: Poirot	53
	6. Programming Examples: Bottom-Up Parsing	61
	7. Co-routining and Parallelism	64
	8. Programming Examples: Inland Spills	67
	9. Programming Examples: Circuit Diagnosis	76
	10. Frames and Arrays	91
	11. Programming Examples: Truth Maintenance	96
	12. A metaProlog Simulator	117
	13. Semantic Foundations	138
	14. Implementation Considerations	146
	References	150

(Parts 1-7 of Part B written with Tobias Weinberg

Part A. Introductory Material and Achievements

Original Abstract

"The maintenance of large volatile knowledge bases is the focus of this project. The viewpoint from which the study is being conducted is that of certain extensions of current logic programming systems, primarily the so-called "metalanguage" systems in which a logic programming language is amalgamated with a portion of its metalanguage. Major thrusts of the work include (1) study of the extent to which such representation mechanisms as frames and semantic nets can be logically treated (thus yielding a measure of independence of representation for the rest of the work), and (2) the use of the "metalanguage" facilities for the maintenance of consistency and integrity under change and other questions of analysis of the knowledge base."

Extended Summary

Computer-based systems to aid human intelligence analysts are instances of a generic class of systems known as *tracking systems*. Such systems minimally consist of a knowledge base in which records representing the analyst's concerns are stored. A useful organization of such knowledge bases distinguishes between events and event-lines. Events are relatively discrete in time, such as signal reports or activity reports, while event-lines are extended, continuous sequences of events. Events may be thought of as discrete points, "plotted" on some event-line. One may also impose a hierarchical structure among event-lines with individual event-lines constituting components of some "higher-level" event-line. For example, a group of event-lines representing individual aircraft flight tracks might constitute a sortie event-line. Note that some individual events, e.g., a particular signal report, may be plotted against the higher-level sortie-line, rather than against any particular aircraft event-line. The first problem to be noted is that of designing appropriate data structures to represent events, event-lines and their relationship in the knowledge base.

The second problem arises from the dynamic character of the knowledge base. New entries (or deletions of existing entries) are steadily made, both by the human analyst and possibly by other computer programs. The problem is to avoid degrading the knowledge represented in

the database via mistaken or inconsistent entries, and to flag "disturbing" or "non-nominal" entries. This is the *mainenance problem*. In systems of more than trivial scope, this will require a maintenance subsystem capable of examining and manipulating the knowledge base.

The third problem is that of assisting the analyst in generating hypotheses and scenarios, and using these to reach conclusions concerning events and event-lines in the knowledge base. These include such problems as whether a given event should be plotted against a particular event-line, or projecting likely extensions of an event-line (i.e., projecting likely events to occur on a given event-line).

This project dealt with basic research directed towards providing a programming system containing powerful tools adequate for the solution of these problems. The project also set out to test these tools in the preliminary exploration of methods of solution for the last problem. The focus of the work was a system called *metaProlog* which belongs to the Fifth Generation family of programming languages. metaProlog is a direct extension of Prolog designed to remedy some of the latters fundamental inadequacies. Prolog's attractiveness for the management of complex knowledge bases lies in its rule-based deductive character. However, ordinary Prolog's facilities for manipulating the databases themselves and for reasoning about them are quite poor and of a non-logical character. The major step on the way from Prolog to metaProlog lies in regarding databases, or *theories* as they are called, as first-class objects, capable of being passed as arguments to procedures and returned as values of variables. This extension provides a very powerful programming tool, useful in constructing data structures for representing events and event-lines in a more flexible manner similar to a generalized notion of frame, while at the same time providing a logically sound method of manipulating multiple database and contexts.

1.0 LOGIC PROGRAMMING

Logic programming utilizes formalized mathematical logic as the basis of programming languages for controlling computers. Its principal practical realizations are the Prolog systems invented by Colmerauer and Kowalski.

1.1 Proofs And Programs

Formal logics, which constitute the basis of logic programming, are concerned with the construction of proofs for assertions.

The overall structure of logic as a computational formalism can be described as follows. A program is produced by constructing a *theory* T together with a distinguised formula A. The theory T constitutes a logical description of the domain in which the computations are to take place (e.g., blood diseases and therapeutic antibiotics, econometric equations for forecasting, messages in an intelligence assistance system). The formula A is an *activation method* for the program; input and output is accomplished by means of the free variables in A. Suppose that A(X,Y) has the free variables X and Y, where X is intended to be used for input and Y for output. Then given any term s describing an input, a logical computation amounts to a search for a term t together with a proof P of the assertion A(s,t). The proof P is based on the theory T. If such a term t and proof P can be found, the computation is said to *succeed* and the term t is its output. If no such pair can be found, the computaion is said to *fail.*

1.2 Procedural Interpretation

The key to the power of logic in programming lies in the existence of two interpretations for formulas A belonging to a theory T. The first, the *declarative interpretation,* views A as describing some property of the entities under consideration. This is the traditional view of logical formulas, and is the interpretation intended when A is used as part of a program specification. The second, the *procedural interpretation*, views A as giving directions for the solution of some problem. This procedural interpretation, devised by Kowalski and Colmerauer, is especially perspicacious for the Horn clause logic systems which include the Prolog systems.

1.3 Representability

The desired amalgamation of object language and metalanguage uses a construction which is a special case of the representation of an intuitive or model-theoretic relation R by a predicate symbol P in the context of a set of sentences (i.e. , a theory).

In general:

> A predicate symbol P *represents* a relation R in the context of a set of sentences T if and only if :
>
> There is a naming relation which pairs individuals i from the domain of R with terms i' of the language of T in such a way that the following holds:
>
> for all $i_1, i_2, \ldots i_n$ in the domain of R,
>
> $(i_1, i_2, \ldots, i_n) \varepsilon R$ if and only if $T \vdash P(i_1', i_2', \ldots, i_n')$.

The symbol $\vdash$ indicates the provability relation. That is, if A is a formula and T is a theory, then

$T \vdash A$

means that:

> There exists a proof of A based on the axioms of T.

Now suppose that R is the provability relation $\vdash_L$ of a language L. (In our intended applications L is the full standard form of logic or some subset such as Horn clause logic.) To represent $\vdash_L$ in another language M (possibly identical to L) it is necessary to name sentences, sets of sentences and other linguistic expressions of L by means of terms of M. In general, if A is a linguistic expression or a finite set of expressions of L we will write either "A" or simply A' to stand for a term of M which names A.

1.4 The Representation Of Provability

Let *demo* be a binary predicate symbol of M, where M functions as a metalanguage for T. If PR is a set of sentences of M, we say that

demo *represents* $|-_L$ *relative to Pr* if and only if

for all finite sets of sentences T of L and all single sentences B of L,

$T |-_L B$ if and only if $Pr |-_M demo(T', B')$.

Recall that T' is the name of T in language M. Note that this notion of representability does ***NOT*** require that the negation of demo,(if it expressible in M) represent unprovability in L relative to Pr. Indeed the undecidability of first-order logic entails that for no representation of provability in L (in a finitary system) does the negation of that representation in turn represent unprovability in L

It is essential to note that nothing in the foregoing definitions forces the languages L and M to be distinct. While our intution is to read these definitions with the assumption that L and M are distinct languages, careful examination shows that the definitions containing nothing *requiring* L and M to be distinct. Thus it is conceivable that L be identical to M. In this special case, the definition of representability would read as follows:

demo *represents* $|-_L$ *in L relative to Pr* if and only if:

for all sets T of sentences of L and all single sentences B,

$T |-_L B$ if and only if $Pr |-_L demo(T', B')$.

We can carry this even further. Since the quantifier "for all sets T" ranges over all sets of sentences of L, and since Pr is one of the sets of sentences of L, we would obtain the following consequence for any language L and theory Pr satisfying the foregoing defintion:

For all single sentences B of L,

$Pr |-_L B$ if and only if $Pr |-_L demo(Pr', B')$.

The existence of such languages is demonstrated by Gödel's famous construction showing the Incompleteness of Arithmetic, as we will discuss in more detail later. It turns out that in general, beginning with any reasonable first-order language L_0, one can extend L_0 to a language L which contains a theory Pr satisfying this definition. In particular, this is true for the languages use for Horn-clause logic, the basis of Prolog. In fact, the basic expressiveness of Prolog-type languages is sufficient to directly construct their own proof predicates, as shown in the following.

The following two clauses D1-2 constitute the top-level of a Horn clause representation of Horn clause provability. (Both the object language L and the metalanguage M are Horn clause logic.) By virtue of the procedural interpretation of Horn clauses, D1-2 can also be regarded as the top level of an interpreter for Horn clause programs.

D1) demo(PROG,GOALS) <— empty(GOALS).

D2) demo(PROG,GOALS) <—
select(GOALS, GOAL, REST),
member(PROC, PROG),
rename(PROC, GOALS, VARIANT-PROC),
parts(VARIANT-PROC, CONCL, CONDS),
match(CONCL, GOAL, SUB),
apply(CONDS & REST, SUB, NEWGOALS),
demo(PROG, NEWGOALS)

1.5 A Database Management Example

Database management requires a combination of object language and metalanguage. The object language is used to pose ground (yes/no) queries against the database. The metalanguage is needed to specify the database, to update and maintain the database as it changes in time and to pose queries which extract useful information from the database. The following top level of a simplified database management system (DBMS) illustrates how the demo predicate can be used to interface the object language and metalanguage.

In this description of a DBMS, the predicate

assimilate(CURR_DB, INPUT, NEW_DB)

describes the relationship which holds when the assimilation of an input sentence into a current database results in a new database (possibly identical to the current one). The terms x & y and x-y name the sets x∪{y} and x-{y} respectively.

A1) assimilate(CURR_DB, INPUT, CURR_DB) <—
demo(CURR_DB, input).

A2) assimilate(CURR_DB, INPUT, NEW_DB) <—
belongs_to(INFO, CURR_DB),
INTER_DB = (CURR_DB - INFO),
demo(INTER_DB & INPUT, INFO),
assimilate (INTER_DB, INPUT, NEW_DB).

A3) assimilate(CURR_DB, INPUT, CURR_DB) <—
demo(CURR_DB & INPUT, false).

A4) assimilate(CURR_DB, INPUT, CURR_DB & INPUT) <—
independent(CURR_DB, INPUT).

The clauses A1-4 respectively deal with the folowing cases:

- A1: The new information is already implied by the database;
- A2: The new information implies information in the database;
- A3: The new information is inconsistent with the database;
- A4: The new information is independent from the database.

Clause A2, in particular, selects one item of information in the current database, removes the item if it is implied by the rest of the database together with the INPUT, and recursively assimilates the INPUT into the smaller database. The constant symbol false names the empty clause, which denotes contradiction. Therefore

demo(T', false)

expresses that T is inconsistent.

The predicate

independent(CURR_DB, INPUT)

can be represented in a variety of ways. The clause

A5) independent(CURR_DB, INPUT) <—
¬ demo(A1-3, "*assimilate(*CURR_DB, INPUT, NEW_DB*)*").

in particular, uses negation by failure to state that the input is independent from the current database if it cannot be assimilated by any of the preceding procedures A1-3.

The clauses A1-5 can be imbedded in a program which processes input streams against the current database. The predicate

process(CURR_DB, INPUT_STREAM, NEW_DB)

describes the relationship which holds when assimilating a stream of inputs into a current database results in a final new database:

P1) process(CURR_DB, nil, FINAL_DB) <— FINAL_DB = CURR_DB.

P2) process(CURR_DB, INPUT.RESTIN, FINAL_DB) <—
assimilate(CURR_DB, INPUT, INTER_DB),
process(INTER_DB, RESTIN, FINAL_DB).

The clauses P1-2 and A1-5, together with the appropriate lower level clauses and the representation Pr of provability, constitute a complete, if somewhat simple-minded, database management system.

1.6 The Amalgamation

We have already noted that object language problems of the form

"Find a proof of B from T in L"

(which we will briefly write as T ?|-$_L$ B) can be replaced by metalanguage problems

Pr ?|-$_M$ demo(A', B').

Consequently, the metalanguage can replace the object language altogether. That is, we could dispense with the object language as an

independent entity, and work in the metalanguage with the names of object language formulas by using demo. On the other hand, many object language problems can be solved more naturally and more efficiently in the object language than in the metalanguage. That is, the proof search meachanism in the object language solves the problems more efficiently than the search for proofs of "demo(...,...)" in the metalanguage. This is because "demo" is a kind of interpreter, while the object language "directly executes" the problems (at least from the relative point of view of comparing the object language and metalanguage proof search mechanisms.) Thus it is desirable to combine the directness of the object language with the power of the metalanguage in an amalgamation which facilitates the communication of problems and their solution between them. Such communication is accomplished by means of the following linking rules:

1) $$\frac{Pr \vdash_M demo(A', B')}{A \vdash_L B}$$

2) $$\frac{A \vdash_L B}{Pr \vdash_M demo(A', B')}$$

These rules simply restate the two parts of the definition of representability. The first rule allows the metalanguage to communicate metalanguage solutions of object language problems to the object language. The second rule allows the object language to communicate the solutions of its problems to the metalanguage. (These linking rules are what Weyhrauch[1980] calls *reflection principles*. The use of EVAL in LISP is also similar to the use of these rules.)

To summarize thus far, M functions as a proper metalanguage for L if the following hold:

a) There is a naming relation which associates with every linguistic expression of L at least one variable-free term of M. (A single expression of L might have several associated names in M. But every variable free term in M is associated with at most one expression of L.)

b) There is a set Pr of sentences of M (involving the symbol 'demo') which

is a representation of $|-_L$ by means of a predicate symbol 'demo' such that the linking rules (1) and (2) hold.

The only restriction on the languages L amd M imposed by this definition is that the metalanguage M be adequate for the representation of the provability relation of L. Horn clause logic is more than adequate to function as a metalanguage for itself. Notice, moreover, that the amalgamation allows the case L=M, where the two languages are identical. This case is of special importance, as it allows the formulation both of sentences which mix object language and metalanguage, and of self-referential sentences.

2.0 LOGIC AND KNOWLEDGE BASES

2.1 Logical View Of Knowledge Bases

The traditional logical views of databases views a database model-theoretically as a particular model *M* of a certain set D of first-order formulas. The logic programming view sees a database proof-theoretically as a theory T whose axioms include D (cf. Nicolas and Gallaire [1978]). In the model-theoretic view, answering a query amounts to truth-functionally evaluating the query over the model *M*. From the proof-theoretic point of view, answering a query amounts to attempting to prove the query in the theory T. This proof-theoretic view seems to solve the traditional problems of null values and incomplete information (cf. Reiter[1981]). For the management of change in volatile complex knowledge bases, the proof-theoretic point of view appears to provide tools to intelligently manage the complexity engendered by complex queries and updates. This is because proofs provide explicit connections between elements of the database, whereas truth-functional evaluation provides no such connections.

2.2 Correctness Of Knowledge Bases

In one way or another, every knowledge base (KB) models some aspect of the "real" world. The correctness of the fit between the knowledge base and the world must be maintained in the face of change in the world (which must be matched by changes in the KB). The minimal constraint to be met is that after each change, the KB remain self-consistent. But this alone is insufficient to maintain correctness, since many changes would leave the KB self-consistent, but no longer correctly representing the intended portion of the real world. Additional constraints, among them the usual sorts of integrity constraints, are needed to control the change. (It is important to note that integrity is a meta-level concept relative to the object language. Integrity constraints are properties which are predicated of object language formulas or theories, and as such are metalevel character.) However, consistency remains the key issue, since the additional constraints are used simply by requiring that the changed KB remain consistent with respect to these constraints. This is part of the problem of *truth maintenance*.

2.3 Truth Maintenance

The core practical problem of truth maintenance is one of efficiency: if a proposed update or new fact contradicts the present knowledge base, this must be discovered in reasonable time. (Of course, deciding what to do about it -- ignore the update or revise the knowledge base -- must also be accomplished efficiently.)

The fundamental aspect of this project's approach to efficient truth maintenance is the utilization of the theory machinery of the *metaProlog* system to record computed proofs and justifications, maintain sophisticated proof-theoretic information about the knowledge base, and express "control" information about how to go about verifying consistency in particular settings. For example, let the knowledge base T_0 be regarded as the union of (in general, non-disjont) consistent theories $T_0 = T_1 \cup T_2$. Suppose A is a formula such that $T \cup \{A\}$ is inconsistent. By the classical Joint Consistency Theorem (cf. Shoenfield [1967]), there must be a formula B of the common language of T_1 and $T_2 \cup \{A\}$ such that T_1 proves not(B) and $T_2 \cup \{A\}$ proves B. That is, we have:

$$T_1 \vdash \neg B \quad \text{and} \quad T_2 \vdash B.$$

If T_1 and T_2 are suitably chosen so that the common language is exceedingly small, the possible forms for B are severely constrained. In the optimal case, B must be a variant of A. But then the search for inconsistency can be restricted from all of T_0 to a search for a proof of not(B) from T_1.

Note that since theories are now regarded as "first-class objects", they themselves can enter into relations in a database. Thus the information necessary for maintenance of the pairs (T_1, T_2) for a primary knowledge base can be represented in rule form in a (secondary) knowledge base.

2.3.1 Deriving Expectations -

The 'theory mechanism' of metaProlog can be used to formulate and maintain "expectations" regarding the knowledge base. These include both static expectations of the sort typified by integrity constraints (e.g., for a particular data relation r(X,Y), the entered values for Y must be integers in the range 50 to 300), and dynamic expectations regarding patterns of

change. For example, if over a given period of time, updates for r(x,Y) have all caused the values of Y to be monotonically increasing in time, the system should "expect" that future updates of r(x,Y) will furthur increase the values of Y. Such dynamic expectations can either be derived by the system or can be included as iron-clad constraints in the basic integrity machinery expressing the "fit" between the knowledge base and the world it models.

When a proposed update contradicts a basic integrity constraint or a derived constraint, the system must react, either questioning the quality of the data involved in the proposed update, or revising the knowledge base to accomdate the update. The basic constraints can have logic procedures attached to them for specification of such reactions. The derived constraints can have their "proofs" attached to them to guide the reaction process.

The project explored "what can be said" about the knowledge base using the facilities of metaProlog. The sorts of "things to say" may include more sophisticated static integrity constraints than can normally be expressed, as well as dynamic constraints. To cater to the expression of dynamic constraints, the knowledge base will almost certainly incorporate temporal references, either through "time-stamping", as suggested by Kowalski[1981], or using of a "validity interval" which is attached to each assertion and which can contain variables at either end of the interval. Beyond the rather elementary dynamic constraint suggested above, the project explored the expression of sophisticated descriptions of expectations for the knowledge base. These can include alternative outcomes depending on events in the world being modelled. A particularly interesting use of such alternatives would be in the construction of alternative schenarios in intelligence knowledge bases, such as those monitoring space missle launches,

2.3.2 Knowledge Base Analyst -

In addition to formulating specific (static or dynamic) constraints, the facilities of the metalanguage can be used to formulate knowledge base analysis rules. For example:

> If a relation has been updated in such a way that all its arguments save one are fixed, and the values of this last argument are increasing numbers, and if the period over which such updates have persisted is at least N time periods, then it is reasonable to expect these increases to continue.

Using this point of view, rule-based "knowledge base analyst" expert systems can be constructed. Such expert systems would be expected to contain "universal" rules applicable to most knowledge bases as well as domain specific rules conditioned by the domain of the particular knowledge base being controlled.

2.3.3 Logic And Knowledge Representation -

The theoretical portions of the project attempted to remain independent of particular knowledge representation choices. Several routes to this end suggested themselves. The first is to regard the machinery of elementary frames and semantic nets as "implementation overlays" on a basically proof-theoretic relational scheme, treating them as sophisticated indexing schemes or storage grouping schemes. The widely used inheritance relations between frames, such as "is-a" or "a-kind-of", appear to be logically treatable by use of the "semantic net as indexing scheme" coupled with the expressive capabilities of the metalanguage. Roughly, if p and q are frames, and if "p is-a q" holds, then:

(1) Logically, the pair (p,q) is a tuple in the is-a relation, but the implementation of "is-a" may be network rather than relational "behind the scene"; and

(2) Logically, p inherits properties from q via the metalanguage rule:

$$(\forall F, g)[\ \text{prop-of}(F, q)\ \&\ \text{name-of}(g, \text{'}F(p)\text{'})\ \&\ \text{is-a}(p,q) \rightarrow \text{demo}(T,g)\ ] \qquad (R)$$

Here "prop-of(F, q)" means that F is a property holding of instances of the generic frame q, "name-of(g, 'A')" means that g is a (metalanguage) name of the formula A, and demo(T, 'A') means that A is provable in the theory T. From a logical point of view, deduction is necessary to use (R) to conclude that F(p) holds. However, the network implementation of is-a extends to a network implementation of (R) to allow the conclusion F(p) to be obtained by fast pointer following.

2.3.4 Virtues Of The Logical Approach -

This logical approach to truth maintenance has several fundamental virtues:

1. The expressiveness of the extended metalanguage appears to allow the complete expression of the necessary constraints on a volatile knowledge base;

2. If the logic system is used not only as a constraint language, but also as the programming language for knowledge base implementation [as well as query language], proving the correctness of a knowledge base implementation becomes a feasable possibility because the gap between constraint and implementation expression is so narrow;

3.0 Project Plan and Achievements

The original project set out to explore these and related theoretical ideas, to attempt to build a prototype extended metalanguage/knowledge base maintenance system, and to exercise it with one or more non-trivial knowledge bases.

Project Staff and Contributors

The following persons were employed as graduate assistants at various times during the project: Hamid Bacha, Aida Batarekh, and Tobias Weinberg. Mr. Weinberg was also employed as a research associate during the second year of the project. He made very substantial contributions to the work.

The following persons, while not directly employed on the AFOSR grant, have made substantial contributions to metaProlog and its applications. Some have been graduate students at Syracuse University who, while supported from other sources, worked on aspects of the project, or are colleagues from other institutions who have contributed by their valuable discussions with us: Kevin Buettner, Ilyas Cicekli, Keith Hughes, Robert Kowalski, Robert Moore, Hidey Nakashima, Andy Turk, Maarten van Emden, and Christopher White.

Original Schedule

Period 1: Theoretical work, including elaborating and working out the approaches listed above; detailed examination of some existing systems; experiments with an existing experimental metalanguage interpreter; preliminary design work on prototype metalanguage system implementation.

Period 2: Continued theoretical work; preliminary design of expert knowledge base analyst; extensions of metalanguage system design to support results of theoretical work; construction of prototype extended metalanguage system.

Period 3: Experiments with expert knowledge base analysts; refinements of design for knowledge base analysts and construction of full prototype analyst; experimental operation of knowledge base maintenance/analysis system using one or more non-trivial knowledge bases.

Summary Of The Work

We extensively, but not exhaustively, surveyed much of the literature, and became convinced that many of the advantages of frames and semantic nets can be captured in logic programming systems by a combination of new storage organizations and relatively minor modifications of the interpreters. In the case of frames, this was fairly well worked out, and one version was originally partially installed in the experimental metalanaguage interpreter coded in DEC-10 Prolog. The technique which was used in this first approach was to organize the storage for the clauses concerned with frames according to the terms to which the predicates apply, rather than according to the predicate being applied, as is the standard technique. This allows the predicates applying to a given term (say their first argument) to be stored together, rather like a generalized record structure. Some of these predicates can be IS-A or A-KIND-OF predicates whose second arguement is another frame. The modifications to the interpreter allow it to take advantage of inheritance along these hierarchies without distorting the logic on the surface of the program. The use of this technique -- grouping information according to the term to which it applies -- is being utilised in a similar way to capture the applications of semantic nets. This approach works quite well and is perfectly logical for static frames. It also allows updates for non-static frames by use of assert and retract, since the frame is represented in the database. For determinate programs, this has a logical basis. However, if a program must backtrack over an update to the frame, ordinary implementations of Prolog backtracking cannot resent the frame. Proper behavior under backtracking can be regained by extending the implementation to support a "backtrackable assert and retract." However, the simplicity of this implementation loses the logical semantics.

We developed a second approach to the implementation of frames based on the use of Prolog's structured terms. These terms are quite like labelled records in conventional programming languages, and are quite appropriate for the representation of frames. Again, this approach works very well for static frames, and requires no change to the basic Prolog implementation. However, to cater to dynamic, updatable frames, a form of

destructive assignment must be utilized. This, of course, is on the face of it highly non-logical. But like our first approach, for determinate programs, it can be given a logical basis. But once again, if the program must backtrack over frame updates, the changes introduced cannot be recovered. However, by extending the system to support what is known as "event trailing", it is possible to both recover the changes and to regain the logical semantics, even in the face of backtracking. We will discuss this further later in this document.

The most promising approach is the representation of frames by means of theories in the metalanguage system, which is discussed extensively in Part B of this document.

We explored the capabilities of our original experimental metalanguage interpreter (written in standard Prolog) and extended it substantially in the process. These extensions went in two directions: Expressing some existing expert systems in the metalanguage, and beginning to build knowledge base management systems. The ultimate goal of the latter of course was systems with extensive truth maintenance capabilities, as described above. In doing this, we developed a powerful technique whose potential has yet to be fully developed. The technique can be briefly described as follows. Starting from the traditional database point of view, we tend to think of a knowledge base as a "flat" item, a collection of concepts and facts about some particular kinds of objects (e.g., diseases and human beings.) Moreover, we tend to visualize truth- and integrity-maintenance mechanisms as "higher-level" entities supervising the development and change of the base-level knowledge. However, once theories have been accepted as first-class entities in their own right, the base-level theory can "talk about" the theories which make up its maintenance system. Thus the base-level theory can contain a predicate

integrityTheory(X)

which "points to" the theory X used to maintain basic elementary integrity of the database under updates. The database manager can be coded so as to look for such a predicate and use the corresponding integrity theory whenever an update to the base-level theory occurs. We have included a simple example of such a database manager, database, and integrity theory later in this report.

In the realm of expert systems, we extensivly explored two systems: a version of the RAND Corporation's ROSIE-based Inland Spills expert for

Oak Ridge, and a fault finder for digital devices based on work of a student (K. Esghi) of Kowalski. The latter has particularly improved our understanding of the needs of the metalanguage system. Both are discussed in detail in Part B of this report.

Work on the very high-level notion of deriving expectations did not proceed as far as we would have liked. This appears to be due to two causes. First, the difficulty in creating a powerful implementation of metaProlog in which to run serious experiments with knowledge bases. (We discuss implementation below.) Second, the lack of well-developed conceptual bases for such "expectations". In part, the latter is due to the former. It is to be expected that as one gains experience with sophisticated knowledge bases built using the sort of conceptual tools found in metaProlog, that intuitions and concepts suitable to the problem will develop. However, from the efforts we have expended struggling with these ideas, we have come to believe that workable notions of expectations will be heavily domain-dependent. Expectations are grounded in the notion of change (with non-change as a special case). Human beings carry with them an immense amount of knowledge concerning which (kinds of) things normally undergo changes and the sorts of possible changes they normally undergo. Expectations are less or more formal predications that change will occur normally. Surpise occurs in one of the following general situations:

1) A normally unchanging thing under goes some sort of non-normal change;

2) A normally changing thing fails to undergo change;

3) A normally changing thing changes in a manner not among it normal collection of possible changes.

[Note that 2) can be seen as a special case of 3).]

If interpreted sufficiently abstractly, 1-3) categorize all things and changes. But because of this and because of the vast scale of the world we perceive, both directly and indirectly, 1-3) are useless in and of themselves. To render them useful, humans add further knowledge to 1-3). Specifically, humans add the ability to categorize situations, and within situations of a given type, knowledge of things normally change, and what the normal range of possible changes of these things will be. [It could be argued that

this latter knowledge is the essence of the ability to categorize situations, but we will ignore the subtlty of this point.] Part of this knowledge would classed as common sense, and much of it as specific to the particular situation, and hence as domain-dependent, with the latter usually being the most useful knowledge. It seems apparent that much of this knowledge is represented by relatively rigid *scenarios* or *schemata* describing the things and changes occuring in the situation. Application of these scenarios to a situation involves simple or complex matching of the situation against the description provided by the scenario.

These scenarios will have greater or lesser generality to the extent that they apply to many or few situations. Those with greater generality tend towards the realm of common sense, while those of lesser generality tend toward the "technical" or domain-specific. Given a description of a situation type, the problem of deriving expectations is that of selecting an appropriate collection of scenarios and determining which of them match the given situation description. Now while simply presenting an adequate situation description is no mean feat, and matching it against a scenario scheme is even harder, both of these tasks appear to be much less difficult than the problem of selecting an appropriate collection of candidate scenarios The present evidence would seem to indicate that human beings do not infer this collection of candidates, but rather that the collection of candidates is directly empirically associated with the situtation type, either through formal instruction or direct experience. Consider the domains of medicine and military or economic conflict. There are no general principles available in medicine which allow physicians to deduce the course of a given medical situation. The scenarios associated by physicians with disease states have been hard-won by the medical community through generations of statistical and clinical observation. Similarly, the schemata wherewith to assess military or economic conflicts and suggest their possible courses cannot be deduced in a principled manner, but is learned through historical study and direct participation in events.

This ability to associate an appropriate collection of scenarios with a given situation is clearly a form of expertise. As such, it will exhibit all the well-known difficulties of acquisition and formalization exhibited by other forms of expertise such as fault diagnosis. In all likelihood, the problem of situation assessment is much worse than that of fault diagnosis, both becuase more information must generally be processed, but also because the conceptual basis of course-of-events description and prediction is much less well developed. (More people can function as successful

diagnosticians, correctly determining the causal basis of a faulty state, than can function as successful prognosticators, correctly determining the possible outcomes of a normally well-described state of affairs.)

4. Students & Publications.

Students

Hamid Bacha	PhD	~	12/87
Aida Batarekh	PhD	~	12/87
Kevin Buettner	MS	~	12/85
Ilyas Cicekli	PhD	~	5/88
Keith Hughes	MS	~	5/86
Andrew Turk	BS	~	12/86
Tobia Weinberg	MS	~	5/84

Publications

Refereed Journals

• K.A. Bowen, *Meta-level programming and knowledge representation,* **New Generation Computing**, v.3 (1985), pp.359-383.

Refereed Conferences

• K.A.Bowen & T.Weinberg, *metaProlog: A metalevel extension of Prolog,* **Proc. 1985 Symp. on Logic Programming,** Boston, 1985.

• K.A.Bowen,K.A.Buettner, I.Cicekli, A.K.Turk, *The design and implementation of a high-speed incremental portable Prolog compiler,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

• K.A.Buettner, *Fast decompilation of compiled Prolog clauses,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

• A.K.Turk, *Compiler optimizations for the WAM,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

Invited Conference Talks

• K.A. Bowen, *Meta-Level Techniques in Logic Programming,* **Proc. Artificial Intelligence '86 Conf.**, Singapore, March, 1986.

• K.A. Bowen, *New Directions in Logic Programming,* **Proc. 1986 ACM Annual Computer Science Conference**, Cincinnati, 1986.

• **Workshop on Meta-Level Architectures & Reflexion** -- K.A. Bowen invited; however, schedule problems prevented participation.

Part B. metaProlog: Design and Application

Contents

1.	Introduction	23
2.	Meta-level Programming and metaProlog	28
3.	The metaProlog System	35
4.	Quantification and Naming: Language Foundations	43
5.	Programming Examples: Poirot	53
6.	Programming Examples: Bottom-Up Parsing	61
7.	Co-routining and Parallelism	64
8.	Programming Examples: Inland Spills	67
9.	Programming Examples: Circuit Diagnosis	76
10.	Frames and Arrays	91
11.	Programming Examples: Truth Maintenance	96
12.	A metaProlog Simulator	117
13.	Semantic Foundations	138
14.	Implementation Considerations	146
References		150

(Parts 1-7 were written with Tobias Weinberg)

1. Introduction

Prolog has many attractive features as a programming tool for artificial intelligence. These include code that is easy to understand, programs that are easy to reliably modify, a clear relation between its logical and procedural semantics and efficient implementations. However, we perceive several shortcomings, chief among them being difficulty in representing dynamic databases (i.e., databases which change in time) and an apparent restriction to backward chaining depth-first search using backtracking. Our intent in this paper is to discuss an extension to Prolog which preserves its attractive features while curing its ills. Before we proceed, let us examine more closely both the advantages and disadvantages of current Prolog systems.

- **Clear, easily understood code:**

Prolog programs consist of assertions and rules. The assertions can be regarded as a database of explicit extensional facts and the rules can be regarded as serving two functions: (i) extending the explicit database by allowing the intensional definition of some data, and (ii) defining derived relations over the primitive data. Both the assertions and rules have a clear logical interpretation which allows the programmer to proceed in a mode which amounts to defining or axiomatizing the relations of concern to the program. Coupled with a reasonable discipline of commentary and self-descriptive names for variables, predicates, etc., this declarative reading makes Prolog programs very easy to understand. On the other hand, the assertions and rules also have a natural procdural interpretation in which the rules describe methods for reducing the search for a solution for one problem to the (joint) solution of other problems, and the assertions describe immediately solvable problems. This interpretation also makes it very easy to understand the computational intent of Prolog programs.

- **Modular, easily modified programs:**

Each Prolog assertion or rule is implicitly governed by a sequence of universal quantifiers binding all of the variables which occur in it. This limits the scope of any Prolog variable to the assertion or rule in which it occurs, and hence there are no global variables in Prolog programs. This introduces a strong modularity in Prolog programs. A new (derived or primitive) relation can be added with impunity since the only variables it can affect are its own, not any of those belonging to existing relations. An existing relation can be modified and the only potential effects are upon those relations which call it, or which call relations which call it, etc. Among other things, this enables the programmer to practice a strong data encapsulation discipline.

- **Well-developed logical semantics:**

The need for a well-developed reliable semantics for programming languages is widely recognized. Since the clauses of a Prolog program are explicitly viewable as logical formulas, Prolog programs inherit on their face the classical semantics of mathematical logic. The discipline of formal logic has developed over the past 2,500 years as the basis of all scientific thought. As such, the semantics of Prolog is quite close to normal human scientific thinking, a definite advantage to the programmer's ability to understand Prolog programs. Moreover, the mathematical machinery for reasoning about collections of logical formulas has been well worked-out over the past 200 years, providing a powerful tool for formal reasoning about (and ultimately certification of) Prolog programs.

- **Efficient implementation:**

Interpreters and compilers for Prolog have been produced which rival the efficiency of LISP interpreters and compilers for comparable code. These systems have been produced during the relatively short 10 year span of Prolog activity and are largly university research efforts rather than full production grade systems with associated program development environments. Commercial grade systems are just beginning to appear, and it is to be expected that further developments, such as optimizing compilers will appear. Many obvious optimizations can be implemented as source-to-source program transformations (e.g. macros). Such transformers or macro processors are easily written in Prolog itself since it is a superb symbol manipulation language.

- **Difficulties in representing dynamic databases:**

As we will discuss more fully in Section 2, many artificial intelligence applications demand facilities which amount to an ability to dynamically manipulate databases. In order to take advantage of the natural deductive machinery of Prolog, the most natural way to represent a database in Prolog is by means of a set of assertions and clauses. However, most Prolog implementations do not provide any method of segmenting the database, much less viewing such databases as first-class objects which can be modified and passed around. To meet this difficulty, almost all implementations of Prolog have provided ad hoc extensions to the basic logic programming paradigm which allow for dynamic modification of the program database by the program itself. But since the database is the program, these facilities have an effect of modifying global variables and data structures. In many cases, this has a catastrophic effect on the first three of the virtues listed above: The program becomes very difficult to understand, reliable modification of the code becomes very hard to accomplish, and the logical semantics is utterly destroyed. Moreover, even execution is affected. Since these dynamic modification facilities affect the program database itself, it is extremely difficult to garbage collect the space which should be recoverable from retracts from the database. (We know of no system which even tries.)

- **Apparent restriction to depth-first search control:**

Standard Prolog implementations utilize top-down depth-first control coupled with chronological backtracking to explore the search space for a given goal, while many artificial intelligence problems seem to demand other search methods. In part, this apparent difficulty is due to a problem in point of view. Pure LISP itself has only left-to-right evaluation and function invokation as available control mechanisms. What one does is to write deductive interpreters which utilize the appropriate control structures. The same is easily done in Prolog. The programming virtues of Prolog listed above make the writing of such interpreters a relatively pleasant task.

The conflict between the logical semantics and the representation of dynamic databases was perceived by Bowen and Kowalski [ ] who proposed a solution based on incorporating portions of the metalanguage of Prolog into the system itself. The immediately relevant consequence of this proposal was that the resulting system provided for multiple, alternative program databases (essentially a notion of context) in a setting which still preserves the logical semantics, yet which provides exactly the tools necessary for the dynamic character of artificial intelligence applications. By preserving the logical semantics, the amalgamation also preserves the practical programming virtues of clarity, modularity, and ease of modification.

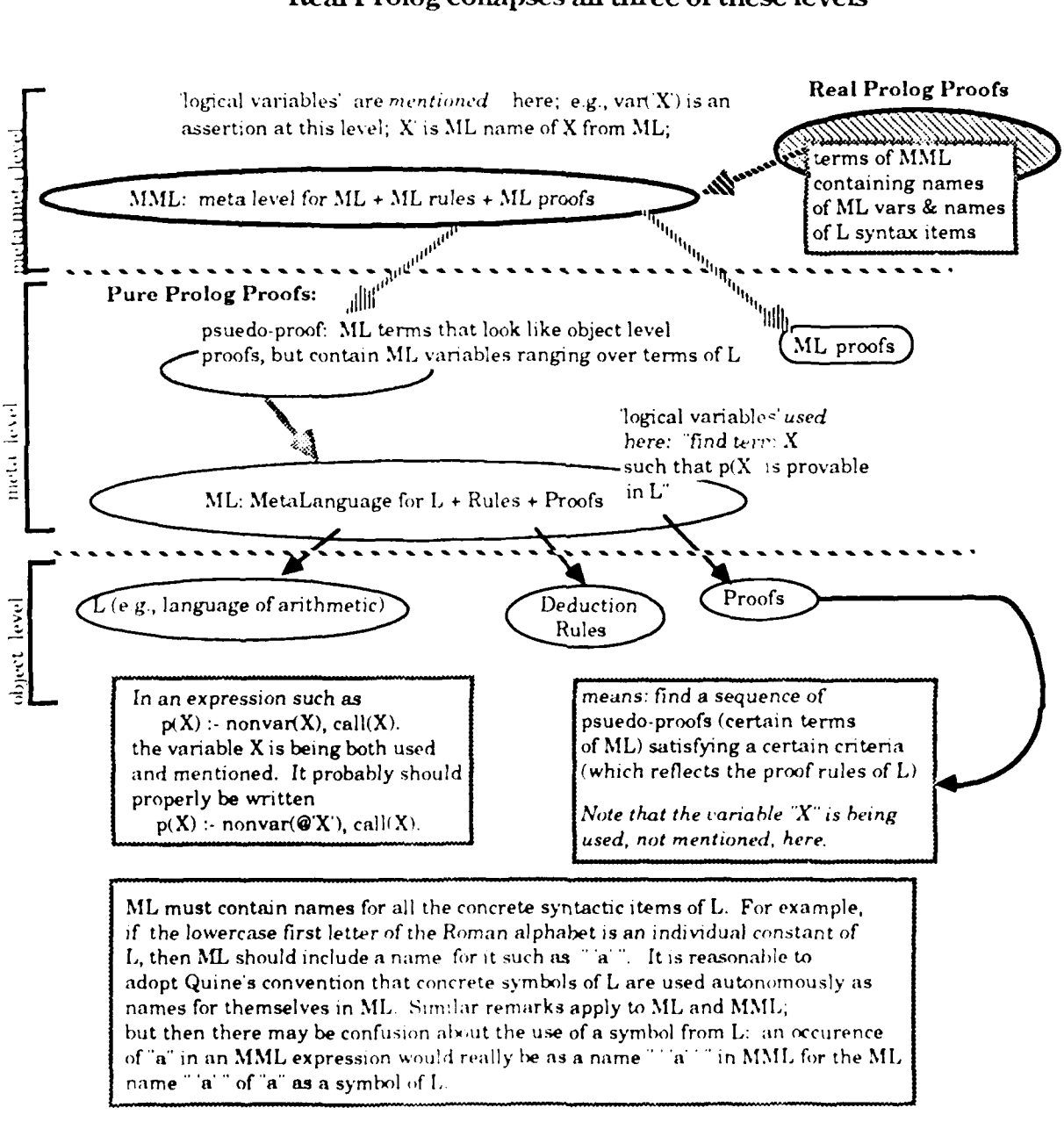
The following sections report on the results of an on-going effort to both develop metalevel programming methods and to build a system which supports them. The system is based on the proposal of Bowen and Kowalski, but goes considerably beyond what they contemplated. After describing the basic outlines of the system, we will illustrate both the approach and the use of the system in several examples, including some which indicate the ways in the which the shortcomings of Prolog mentioned above are overcome.

2. Meta-level Programming and metaProlog.

It is important to make clear our notion of meta-level programming. Our point of view stems from that of classical logic. Early on in the study of language and reason, it was discovered that the distinction between use and mention of linguistic entities was crucial, and this developed into the object-level/meta-level distinction of current mathematical logic. Briefly, one distinguishes between the formal language being used to conduct some (unspecified) axiomatic investigation (the object language) and the language used to carry on any discussion about the object language (the metalanguage). In the full setting of traditional mathematical logic, the metalanguage must be powerful enough to discuss not only the syntactic properties of the object language, but also the semantic (set-theoretic) structures used in interpreting the object language. However, for many purposes (including those of this paper), the metalanguage need only be powerful enough to discuss the combinatorial syntactic properties of the object language. The essential point is that the relations of the metalanguage are about the syntactic entities of the object language: the variables of the object language range over various syntactic entities of the object language. In contrast, the variables of the object language either have no specified range (when it is viewed as a formally uninterpreted language) or range over the entities (possibly extremely mathematically complex) of some specified set (when the object language is treated as being interpreted).

Properly viewed, an ordinary Prolog interpreter is already a meta-level object. For a Prolog interpreter is a particular kind of theorem-prover, and theorem-provers are meta-level entities. The object level consists of a formal logic which is usually (a fragment of) ordinary first-order logic. This fragment is made up of a language and proof predicate. The latter describes which formulas of the language are consequences of sets of other formulas of the language. Most proof relations are composed from more primitive immediate consequence relations. This is the case for the proof relations used in theorem-provers. The meta-level of a theorem-prover is concerned with the manipulation of sets of object-level formulas in the search for a collection of formulas which witnesses the derivability of a given goal formula from a given set of axiom formulas. The prover proper is a meta-level object because its variables range over formulas (and other syntactic classes) of the object level language. The nature of the prover (i.e., the structure of its algorithms)

is obviously dependent upon the nature of the immediate consequence relations of the object level formal logic together with the nature of the allowable methods for composing these relations into proof relations (i.e., the object level deduction rules). Figure B.2.1 illustrates the situation.



The MetaLevel Structure of Real Prolog

Figure B.2.1.

The formal logic constituting the object level of Prolog is the fragment consisting of the Horn formulas together with the Resolution Rule as its sole rule of immediate consequence. The Horn formulas have the forms

$$(\forall x_1)...(\forall x_n)[A_1 \ \& \ ... \ \& \ A_m \rightarrow B] \tag{2.1}$$

$$(\forall x_1)...(\forall x_n)B \tag{2.2}$$

where B and the A_i are atomic formulas and all of their variables occur among $x_1,...,x_m$. It is important to note that this is a formal logic and hence that its variables are uninterpreted: there is no pre-selected domain over which they range. The goal formulas proved by a Prolog system have the form

$$(\forall y_1)...(\forall y_n)[C_1 \ \& \ ... \ \& \ C_m], \tag{2.3}$$

where the C_j are atomic formulas. The first step taken by a Prolog interpreter (or theorem-prover) is to replace the existentially quantified object-level variables of (2.3) by existentially quantified meta-level variables, thus:

$$C_1(Y_1,...,Y_k) \ \& \ ... \ \& \ C_q(Y_1,...,Y_k), \tag{2.4}$$

where $C_i(Y_1,...,Y_k)$ indicates the result of replacing the occurrences of $y_1,...,y_k$ in C_i by $Y_1,...,Y_k$, respectively. Two points are worthy of note here. First, while the object level variables y_j are uninterpreted — *have no specified domain over which to range* — the meta-level variables Y_j are interpreted — *they range over the domain of (syntactic) terms of the object language*. Second, the quantification in (2.3) takes place locally in that formula, whereas the quantification of the variables Y_i occurring in (2.4) takes place globally in the body of the algorithm for the Prolog interpeter. In effect, the Prolog interpreter takes a constructive approach to its attempt to prove (2.3): it will attempt to find concrete terms replacing $Y_1,...,Y_k$ for which (2.4) is provable.

The manner in which the axiom formulas (2.1) and (2.2) are employed reflects the Prolog interpreter's reliance on the Resolution Rule of inference. At any time before completion of the deduction (or abandonment of the attempt), the interpreter has before it a current goal

of the form (2.4). First it selects one of the C_i as the next subproblem to be treated. Secondly, it searches the program or database for a formula of one of the forms (2.1) or (2.2) such that the predicate and number of arguments occurring in B are the same as those of the selected C_i. Third, it strips the quantifiers $("x_1)...("x_n)$ from the formula (2.1) or (2.2), generates new meta-level variables $X_1,...,X_n$, and replaces the variables $x_1,...,x_n$ in (2.1) or (2.2) by $X_1,...,X_n$, respectively, *yielding*

$$A1(X_1,...,X_n) \,\&\, ... \,\&\, Am(X_1,...,X_n) \rightarrow B(X_1,...,X_n) \tag{2.5}$$

or

$$B(X_1,...,X_n) \tag{2.6}$$

Fourth, it attempts to match $B(X_1,...,X_n)$ with C_i using the Unification Algorithm (which may cause binding of various meta-level variables). If this attempted match succeeds, the interpreter modifies the current goal:

> If the matching formula was (2.6), the subproblem C_i is simply deleted, while if the matching formula was (2.5), C_i is deleted and is replaced by
>
> $$A_1(X_1,...,X_n) \,\&\, ... \,\&\, A_m(X_1,...,X_n) \tag{2.7}$$

All of this is work that takes place at the meta-level, being syntactic manipulation of object level formulas. If the attempted match by unification does not succeed, the intepreter seeks alternative formulas (2.1) or (2.2) to the selected one from the database. If at any time, the current goal becomes empty (i.e., all atomic formulas or subproblems have been deleted by matches against formulas of the form (2.6)), the original attempted deduction of (2.3) from the program *succeeds*. On the other hand, if at some point no matching formulas (2.1) or (2.2) for the selected subproblem C_i can be found, the interpreter backtracks, undoing variable bindings and subproblem replacements, and explores alternatives to choices it made previously in selecting matching formulas from the database. If it ever backtracks all the way to the original goal (2.3), it quits, concluding that (2.3) cannot be deduced from the program database, and the attempted deduction *fails*.

Thus a Prolog interpreter really defines a metalevel (or syntactic) relationship between sets of formulas (the program database) and goal formulas, namely the relation that the goal formula is deducible from the program database. However, as commonly implemented, pure Prolog interpreters essentially incorporate the program database as a fixed part of the interpreter, and thus really define a parameterized set of a unary predicates applying to goal formulas. The fundamental operator of standard Prolog systems is thus a one-place operator (usually written **call(...)**) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the proposal set forth by Bowen and Kowalski (1982) was to utilize a system implementing the full deducibily relation described above. Such a system would have metavariables which not only ranged over formulas and terms, but would also allow the metavariables to range over sets of formulas (called *theories*). The fundamental operator of such a system is a two-place operator, usually written **demo(___ , ...)**, which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

In such a system, the only analogue of the standard Prolog database is the global database containing system built-in predicates. All other databases are the values of Prolog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the system predicate **demo(___,...)**, the system predicates include **addTo(___ , ___ , ___)** and **dropFrom(___ , ___ , ___)** which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause (formula (2.1)) containing calls of the form

..., addTo(T1, A, T2), demo(T2, D),... (2.8)

where the theory which is the value of T2 has been constructed by the earlier calls. The effect of (2.8) would then be to construct (in an efficient manner) a new theory T2 resulting from T1 by the addition of the formula A as a new axiom; then the system invokes a search for a proof of the formula D from the theory T2. Since demo implements the proof relation, such programs as (2.8) preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of demo are expressed by what were called *linking rules* by Bowen and Kowalski:

If demo(T, A), then A is derivable from T. (2.9)

If A is derivable from T, then demo(T, A). (2.10)

These rules provide the justification for the implementation of calls on demo in the abstract metaProlog machine as context switches. In essence, at most times the machine behaves as a standard Prolog machine with the current theory (the analogue of the usual fixed program database) indicated by a register. When a call demo(T, A) is encountered, the database (theory) register is changed to point to T and a new search for a deduction of A is begun. Thus the efficiency of standard Prolog computations is preserved and the overhead of meta-level computation is localized in the construction of new theories from old. This method of reflection has been utilized heavily in constructing the abstract metaProlog machine on top of a primitive storage management machine. This approach provides a meta-level programming methodolgy suitable for constructing other methods of exploring the search space of derivations of A from T besides the top-down depth-first approach of standard Prolog. Exploitation of this approach will ultimately provide the meta-level programmer with a library of search strategies which can be (programmatically) invoked depending on the particular problem and context.

Many AI applications require the production and consumption of large data structures, for example lists, and many of these can be of a size which strains or exceeds the resources of the hardware. However, it is often the case that the entire data structure need not really be constructed in its entirety. Rather, the consumption operation could process it piecemeal, either ultimately in its entirety, or even only partially (as in the cases involving search for a component with a particular property). To provide for such a style of programming, we have explored two constructs which allow for the description of co-routined production and consumption processes:

enumerate(Template, Goal, Result) (2.11)

and

streamOf(Goal, Result) (2.12)

From the abstract point of view, enumerations and streams are

first-class objects equally on a par with terms, formulas, and theories. The actual representations of enumerations and streams are as virtual lists which can be potentially infinite. Thus in both enumerations (2.11) and streams (2.12), Result is logically a list which may be extended as consumption processes attempt to access its tail. In the case of enumerations (2.11), Result is the stream of instantiations of Template in the environments corresponding to successful solutions of Goal. Thus it is simply a "lazy" version of the usual Prolog setof operator with the variation that if there are no solutions at all of Goal, (2.11) succeeds and binds Result to the empty list. In (2.12), Goal is expected to be a determinate tail-recursive predicate which constructs the list Result. Intuitively, at each recursive step, Goal places another element on the list; the system evaluates this lazily, suspending furthur action on the recursion of Goal until some consumer attempts to access the (as yet undeveloped) tail of Result.

In both constructs above, it is desireable that Goal be allowed to contain stream variables from other such calls with the consequence that determinate and-parallelism be implemented in the system. To allow the system to easily get this straight, such streamOf and enumeration calls must be wrapped in the "simultaneous" operator construct, as for example:

simultaneous{streamOf(G, L), streamOf(H, K)} (2.13)

where presumably L occurs in the goal H, and K occurs in the goal G. In the present project, the and-parallelism was implemented on sequential hardware in a rather standard co-routining method amounting to time-sharing of the abstract metaProlog machine by the cooperating calls. The design of the abstract machine, however, would allow the cooperating calls to be executed on separate (even heterogeneous) processors in a multi-processor environment (whether tightly coupled or loosley distributed on a network). We discuss this in more detail in Section 7.

3. The metaProlog System

The metaProlog system is designed to be syntactically compatible with the Edinburgh system, to preserve the standard logical semantics, and to incorporate the full two-place proof relation. Thus, while syntactically quite similar to Edinburgh Prolog, metaProlog provides a quite different set of built-in meta-level predicates and allows the metavariables greater range than the Edinburgh system.

There is one major syntactic difference between the two systems. For reasons which will be made clear when we discuss quantification, we require that the implicit universal quantifiers on clauses be made explicit. Thus, for example, the Edinburgh clause

```
append([Head | Tail], RightSeg, [Head | Result_Tail]) :-          (3.1)
    append(Tail, RightSeg, Result_Tail).
```

would be written

```
all [head, tail, rightSeg, result_Tail] :
    append([head | tail], rightSeg, [head | result_Tail]) :-      (3.2)
    append(tail, rightSeg, result_Tail).
```

If the clause contains only one variable, the list brackets in the quantifier can be dropped. Additionally, we allow the programmer to optionally use <— instead of :-, and to connect the literals in the body of a clause by & instead of comma. Thus the Edinburgh clause

```
p(X) :- q(X),r(X).                                               (3.3)
```

might be written

```
all x : p(x) <— q(x) & r(x).                                     (3.4)
```

An expression which contains no meta-variables (but may have object-level variables occurring listed in the quantifer) is called a *closed formula*. A *theory* is either the *empty theory* (designated by **empty_theory**) or is of the form

```
A & U                                                            (3.5)
```

where A is a formula and U is a theory. A theory is a *definite theory* if all of its formulas are closed. (Note that quantified variables may be present in definite theories.) Theories contained freely occurring logical variables are called *indefinite theories*. [Whether or not the programmer is allowed to directly write free logical variables is a matter of design controversy. But indefinite programs can always be created by programs at run-time.]

The proposed built-in predicates of metaProlog include most of those of Edinburgh Prolog with the exception that all those concerned with the "program database" are excluded. Instead, a two-place demo, a three-place demo, the two three-place predicates addTo and dropFrom, the binary predicate axiomOf, and the unary predicate current are to be built-in predicates of metaProlog. Additionally, the two three-place predicates setOf and streamOf, together with the predicate simultaneous are added. These latter three built-in predicates will be discussed in Section 7. The specifications of the former predicates follow.

demo(Theory, Goal)

This call invokes a subsidiary (Prolog) computation which attempts to derive **Goal** on the basis of the program **Theory**. If **Theory** or **Goal** are are not fully instantiated, meta-variables occurring in either may be bound if a successful computation can be found.

Calls on demo support a convenient idiom for describing implicit unions of theories. Specifically, a call of the form

demo(Theory1 & Theory2, Goal) (3.6)

is logically equivalent to the call

demo(Theory3, Goal) (3.7)

where Theory3 is the ordered union of Theory1 and Theory2 in the following sense: If theories are regarded as the ordered list of their axioms, then Theory3 satisfies

append(Theory1, Theory2, Theory3). (3.8)

However, the system does not physically create Theory3, but regards the

expression Theory1 & Theory2 as a description of a *virtual theory*. In effect, when searching for a rule or fact to apply to a selected subproblem of the current goal, it first searches Theory1 for a candidate, and only on failing to find such a candidate in Theory1, it then searches Theory2. Another usage supported is the explicit indication of the axioms of the theory. Namely, if it is desired to search for a deduction of G from A1,...,An, this is achieved by the call

demo([A1,...,An], G). (3.9)

(Internally, theories are represented in several forms. The simplest is just that of a list of axioms, with no special indexing. Retrieval from such a theory amounts to a linear search of the list. Thus, for all but relatively small theories, computation from such a theory will be intolerably slow.)

The two usages can be combined, as in the calls:

demo([A1,...An] & Theory2, Goal)

demo(Theory1 & [A1,...An], Goal). (3.10)

demo(Theory, Goal, Control)

This call causes a subsidary (Prolog) search for a deduction of **Goal** from **Theory** where, however, the search may be modified by information supplied in the control expression **Control**. As with the binary demo, if **Theory** or **Goal** contain uninstantiated meta-variables, these variables may be bound by a successful computation, though some control expressions may cause such meta-variables to remain uninstantiated even after a successful completion of such a call. As with theories, control expressions may be combined, as in the call

demo(Theory, Goal, Control1 & Control2). (3.11)

In effect, a call on the three-place demo first analyses the control information, causing various registers to be set, and then invokes the machinery corresponding to the two-place demo. Initially, the system will support the following control expressions:

depth(N)

This control expression sets a depth limitation of N on branches of the search tree. If a computation exceeds this depth, the branch is failed and backtracking occurs.

proof(P)

This control annotation causes the system to accumulate a representation of the proof branch in the (uninstantiated) variable **P**, allowing the programmer to extract a successful proof for furthur processing, such as providing explanations, etc.

branch(P)

This control expression causes the call

demo(Theory, Goal, branch(B)) (3.12)

to succeed in all cases, binding the uninstantiated variable **B** to the left-most branch of the search tree. Note that in the case that the left-most branch is theoretically infinite, the call will still succeed due to depth bound limitations of the system. Backtracking into this call will cause **B** to be bound to successive branches of the search tree. As discussed in Section 7, the call

setOf(B, demo(T, G, branch(B)), Branches) (3.13)

would cause **Branches** to be bound to the (lazy) list of all branches of the search tree for **G** relative to **T** in the order that they are explored by the system.

bottom_up

Use of this control expression allows a limited amount of single-step bottom-up processing to take place. The call

demo(T1, reduct(T2), bottom_up(T3))

constructs the *reduction* **T2** of **T1** by **T3**. By definition, **T2** will consist of all *reducts* R' by **T3** of rules belonging to **T1**, where if R is a rule with head H and body B, the *reduct* R' of R by T is a rule whose head is H and

whose body B' is obtained from B by resolving some literal of B with some fact in T.

user_choice

Use of this control expression allows the user to specify the choice function for selection of the subproblems during the main loop of the interpreter in the style of Pereira [ ]. Consider the call

demo(T, G, user_choice). (3.14)

The system will expect to find in **T** an assertion

user_choice(UC) (3.15)

as an axiom, where **UC** is a (small) theory containing clauses defining the predicate

choose(Goal, SubProblem).

Here **Goal** will be a representation of the current system goal and **SubProblem** will be bound (when this is run) to the selected subproblem from this goal. When running (3.14), at each cycle of the search for a proof of **G** in **T**, the system will run the subsidiary problem

demo(UC, choose(Goal, SubProblem, Remainder)) (3.16)

with Goal bound to the current goal in order to select the next **SubProblem** and to indicate the **Remainder** of the **Goal** left after this selection. Use of this facility will be illustrated in Section ##.

confidence(C)

While not properly a control description, use of this expression allows for the propagation of confidence values or certainties of the programmer's design. The system provides general methods for attaching terms representing confidence factors to rules and assertions of theories, and for designating predicates for carrying out the propagation of these factors during deduction, as follows. If R is a rule or fact intended to to appear in theory T with confidence C, insertion of the expression

C :: R.

in the source file used to generate T will cause the rule R to be recorded in T and in addition, causes the fact

'$confidence'(R, C)

to be recorded in T. [The actual implementation differs somewhat from this.] Such rules with confidence can also be used in the built-in addTo described below. Assume that normal Prolog control is being used. Operation of the system with the call

demo(T, (A1 & A2 & ... & An), confidence(C))

proceeds as follows. Assume that A <— B is recorded in T with confidence Cr, that A matches A1 via the substitution S, that the result of applying S to B is B', and the result of applying S to A2 & ... & An is G''. The system first solves the goals

demo(T, B', confidence(CB))

and

demo(T, G'', confidence(C')).

Then the system solves the goal

demo(T, '$confidence'(Confi))

and finally solves the two goals

demo(Confi, '$imp_prop'(CB, Cr, CA)

and

demo(Confi, '$cnj_prop'(CA, C', C)).

Thus the programmer is expected to supply (as a sub-theory of T), a theory Confi in which he or she defines the predicates '$imp_prop' and '$cnj_prop' for propagating confidence factors.

This completes the list of currently contemplated control expressions.

addTo(Theory, Clause, NewTheory)

If **Theory** is bound to a theory and **Clause** is a (closed or partially instantiated) formula and **NewTheory** is an uninstantiated variable, this call causes **NewTheory** to be bound to a theory obtained from Theory by adding Clause as a new axiom. What actually happens is that the orignal theory bound to **Theory** is physically modified by the addition of **Clause**, providing fast access to the **NewTheory**. The variable **Theory** is rebound to an internal representation of the result of dropping **Clause** from the theory now bound to **NewTheory**, in a manner inverse the the common method of representing arrays in logic. Thus the original theory bound to **Theory** is still logically available via **Theory**, but access to it is a bit slower. If **Theory** is not bound to a theory or if **NewTheory** is instantiated a run-time error occurs. Note that, unlike the treatment of assert in conventional Prolog, metavariables occurring in **Clause** are NOT converted to universally quantified object variables in the assert fact or rule. This point will be discussed more fully in Section 4 below.

addTo(Theory, Clause, NewTheory, Pointer)

This call is similar to the three-argument form of addTo. Here **Pointer** should be an uninstantiated variable. Running of this call will cause have the same effect as the three-argument form of addTo with the additional effect that **Pointer** will be bound to a representation of an internal pointer to **Clause** as an element of **NewTheory**. As will be discussed below, the value of pointer can be regarded as a meta-level name of **Clause**.

dropFrom(Theory, Clause, NewTheory)

Under the restrictions of addTo, this call causes **NewTheory** to be bound to the theory resulting from the deletion from **Theory** of the first occurrence of an axiom of **Theory** which matches **Clause**. The internal representations and run-time errors are similar to those for addTo.

axiomOf(Theory, Clause)

This call succeeds if **Clause** has been recorded as an axiom of **Theory**.

If **Clause** is uninstantiated, it will be bound to the first axiom of **Theory**. Backtracking into this call will cause **Clause** to be successively bound to the axioms of **Theory**. **Theory** must be bound or a run-time error occurs.

axiomOf(Theory, Clause, Pointer)

This call succeeds if **Clause** has been recorded as an axiom of **Theory** and **Pointer** is a representation of an internal pointer to **Clause** as an element of **Theory**. **Theory** must be bound or a run-time error occurs. Either **Clause** or **Pointer** or both may be unbound, as in the two-argument version of this predicate.

current(Theory)

This call is equivalent to the Prolog definition

```
demo(Theory, current(X)) :-
    X = Theory.                                    (3.17)
```

Note that axioms of the current theory can be directly accessed via the goal

```
<—current(Theory), axiomOf(Theory, Clause).        (3.18)
```

consult(<theory>,<file>) and **consult(<file>)**

If <theory> and <file> are both constants and <file> is an appropriate operating system file name, the first call causes the clauses listed in the file to be added to the end of the theory currently recorded under the name <theory>. If <theory> has not yet been established, it is taken to be the empty theory. The second call is equivalent to consult(<file>, <file>).

Other meta-level built-ins will be described in succeeding sections and in the system manual when it becomes available.

4. Quantification and Naming: Language Foundations

Subsection 4.1: Godel's Reflection Construction

In Section 2 we presented our basic point of view regarding the distinction between object language and metalanguage. In particular, we pointed out that the metalanguage must contain names for all the various syntactic entities of the object language as well as variables to range over those entities. As presented there, it would appear that there must always be a sharp distinction between object language and metalanguage. Certainly this is not the case for natural languages such as English, in which one can carry on discussions of the language in itself. That it is also not necessarily the case for formal languages can be seen by first considering the classic constructions of Godel utilized in his proof of the incompleteness of arithmetic. (Godel carried out his original proof in the context of Russell and Whitehead's Theory of Types; we will be content with a version recast in ordinary first-order logic.) The object language for this construction is simply a version (almost any will do) of arithmetic axiomatized in standard first-order logic, say as presented in Chapter 1 of Shoenfield [1967]. The metalanguage, while not usually precisely specified, is any language containing names for all the primitive symbols of the object language and having the ability to represent concatenation and other primitive syntactic operations; variables whose range includes the syntactic entities of the object language are included. The situation is represented schematically in Figure B.4.1.

The technical heart of Godel's proof lay in showing that the roles of L and M could be essentially reversed (intuitively, that the figure could be inverted). Specifically, Godel showed that L could function as metalanguage for a sufficiently large enough part of M so as to include that portion of M actually used in discussing the syntax of L. The trick lay in showing that natural numbers, the entities which are the intended ranges of the variables of L, could be (in a systematic way) used as names of the syntactic entities of M, and that, given this representation of the syntactic primitive symbols of M, the basic syntactic relations of M could be defined arithmetically in L. Schematically, the situation would then appear as in Figure B.4.2.

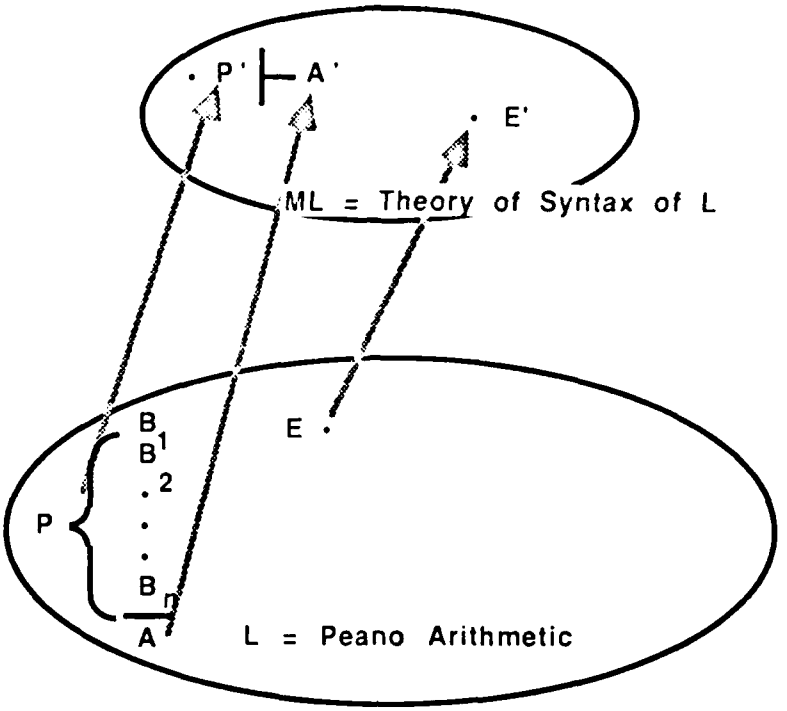


Figure B.4.1. Language and MetaLanguage.

The essential point is that via this reflection through the metalangauge, L has the capability of functioning as its own metalanguage: Numbers can be viewed in and of themselves, or as names of syntactic elements of L; and relations may be simply relations among numbers in and of themselves, or may be seen as relations between syntactic elements of L. In particular, one could define (primitive recursively) in L the proof relation for L itself:

proof(t, f, a) is derivable in L

if and only if (4.1)

t is a number naming a (finite) theory t' in L, f is a number naming a formula f of L, and p is a number naming a finite sequence of formulas p' of L such that p' contitutes a formal proof of f relative to the theory t'.

Smullyan [ ] has provided constructions of formal languages in which this sort of self-reference is direct without need of the intermediate reflection through an external metalanguage. Indeed, even Godel's original construction can be viewed as providing directions for building

the name relation directly.

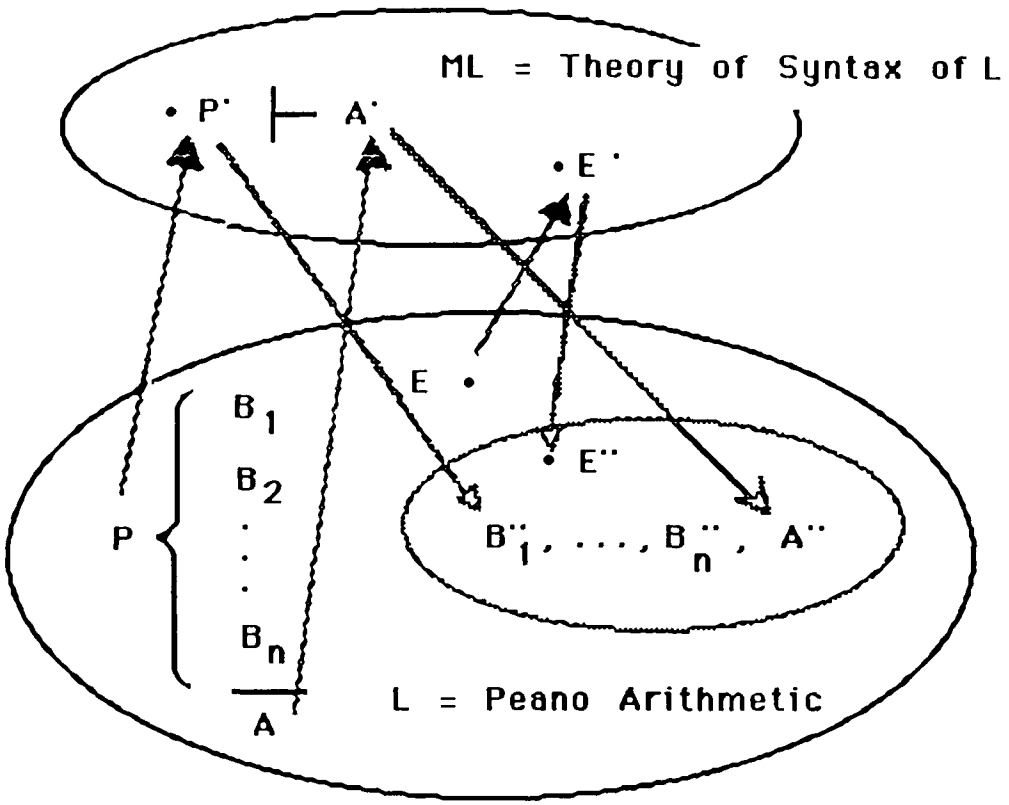


Figure B.4.2 The Metalanguage Reflected in the Object Language.

Abstracting from this discussion, we can see that the two essential requirements we must impose on a language L are:

(i) For every appropriate primitive syntactic entity E of L, there exists a constant e in L naming E;

(ii) There exist relations in L connecting the names of the components of a compound syntactic entity of L with the name of the entire compound entity. [We will elaborate on these requirements below.]

Before proceeding to set forth the formal language for the metaProlog system, we must entertain some considerations on quantification and

the naming of entities.

Subsection 4.2: Quantification

The logical interpretation of Prolog's theorem prover stipulates that variables actually occurring in the program's clauses are in fact implicitly universally quantified object level variables, even though they are syntactically indicated by metavariables. In using a clause, the interpreter replaces these universally quantified object level variables by existentially quantified meta-level variables. The syntactic conflation of object- and meta-level variables is acceptable for pure Prolog deductions, but causes difficulties as soon as assert (and retract) are added to the system. If the expression A contains a metavariable X which is uninstantiated at the run-time execution of assert(A), there is a natural sense in which the call assert(A) is incoherent: the formula to be added to the database is not fully specified. The Prolog approach to this problem is to once again conflate the existentially quantified metavariable X with a corresponding universally quantified object-level variable, actually asserting (all X)A into the database. But there are difficulties in this approach, since it destroys the logical semantics of clauses in which such calls occur. Assuming no clauses for p are in the database, the following two goal statements should be logically equivalent:

:- X = b, assert(p(X)), p(d).

:- assert(p(X)), X = b, p(d).

But the first fails, since it only adds p(b) to the database, while the second succeeds, since it adds (all X)p(X) to the database. To avoid such difficulties, the metaProlog system requires that programmers be explicit about their intentions, clearly indicating universally quantified object variables. Thus, to add (all X)p(X) to a theory T, one would write

addTo(T, (all X : p(X)), U).

Note that if the expression addTo(T, p(X), U) occurs in a metaProlog program clause, X is either a constant or is explicitly universally quantified by a quantifier on the clause containing this call to addTo. In the latter case, on entry to the clause, X is replaced by an existentially quantified variable at a level meta to the clause. If it is not instantiated when the call to addTo takes place, no change to X takes

place. Rather, the formula is viewed as partially instantiated, and the resulting theory U is also seen as partially instantiated.

Subsection 4.3: Naming

In order for any language M to serve as a metalanguage for another language L, M must contain names for all the appropriate syntactic entities of L. Thus, since metaProlog is to serve as its own metalanguage, it must contain names for all of its own syntactic entities, just as any natural language has the ability to name all its own syntactic constructs, both by description and by quoting. To this end, in metaProlog, constants act as names of themselves. For ground items other than constants, metaProlog may provide structural or non-structural names (and sometimes both). The former are compound terms whose structure reflects the syntactic structure of the syntactic item they name. The latter are analogous to proper names in natural languages. In particular, database references (relative to individual theories) act as non-structural (proper) names of the clauses or other theories to which they point. Facilities for manipulating names should be provided, for example, methods of obtaining a (compound) name of a compound expression from names of the expression's components, as well as methods for moving between a name and the thing it names analogous to univ (=..) of ordinary Prolog.

Subsection 4.4: Formal Language Specification

We now will proceed to set forth a formal definition of the current design of the metaProlog language. The first step will be to precisely specify the purely linguistic component which we will refer to as L(mP). Later, in Section 12, we will specify the computational component operationally as a purely formal mathemetical system. Note that these definitions are phrased in a language (technical English) functioning as a metalanguage for L(mP); eventually we will show that mP has sufficient power to act as its own metalanguage. As discussed in general above, this metalanguage (technical English) must have syntactic variables which will range over the various syntactic constructs of L(mP). The range of the (meta)variables of the technical English metalanguage must include the logical variables of L(mP). The logical variables of L(mP) themselves will turn out to function as metavariables for portions of the L(mP) language. We will use expressions of the form <....> to indicate these syntactic metavariables of the technical English metalanguage. Because of this multiple use of meta-levels, including the fact that the

system mP is intended to be able to function as its own metalanguage, the possibilities for confusion are rampant. We caution the reader that careful consideration of the location of the definitions with regard to the definition's presence in mP or in the technical English metalanguage of mP is extremely important. The heavy formalism of this section has been an important tool for elaborating the system and clarifying the distinctions. The use of the formalism is also motivated by the governing desire to provide a system which can be seen to possess a logical semantics.

Definition 4.2. The language L(mP) is specified as follows:

(.1) Any identifier beginning with an alphabetic character is a *constant* (irrespective of whether the initial character is upper-case or lower-case).

(.2) Any sequence of characters beginning and ending with a single quote is a *constant*. Single quotes themselves can be embedded in such a constant by the standard device of repeating them at the point at which they are to occur.

(.3) Any number (integer or real) in fixed or floating point notation is a *constant*.

(.4) Any constant is a *name* of itself.

(.5) There is a countable collection of symbols distinct from all the constants and punctuation of mP ; the elements of this collection are called *logical variables*.

(.6) Any identifier beginning with an alphabetic character may be used as a *functor symbol* (irrespective of the case of the initial character).

(.7) Any functor symbol is a *name* of itself.

(.8) Any constant or logical variable is a *term*. The *principal functor* of a constant is itself; logical variables have no principal functor.

(.9) If <f> is a functor symbol and <t1>,...,<tn> are terms, then

<f>(<t1>,...,<tn>) (4.3)

is a *term*. The functor symbol <f> is called the *principal functor* of the term.

(.10) For every term there is a constant of L(mP) which is a *name* of the term.

(.11) There are distinguished constants:

empty_theory true demo instance meta_name

among the constants of L(mP).

In a fundamental sense, all of the linguistic expressions of L(mP) are terms. The definitions which follow effectively single out terms of special forms to function as specialized syntactic items such as literals, clauses, theories, etc.

Definition 4.4. The *reserved symbols* of L(mP) are the following. (Note that the single quotes are part of the technical English metalanguage -- they are part of its machinery for naming syntactic items.)

':', all, 'if', '<—', ':-', '&', ',', '(', ')', '|', '[', ']', '{', '}', '.', '[]', '()', '{}', '::'.

Definition 4.5. Any term whose principal functor is not a reserved symbol can be a *literal*, including logical variables. When a term is used as a literal, its principal functor is called a *predicate name*; a literal consisting of a logical variable alone has no principal functor, and hence possesses no predicate name.

Definition 4.6. The class of *list expressions* (or briefly, *lists*) is defined recursively as follows:

(.1) [] is a list, called the empty list;

(.2) If <L> is a list and <T> is any term, then

'[]'(<T>,<L>)

is a list.

The formal definition of the language renders every syntactic object either a constant or a term which is written in prefix notation. Human readability and ease of use requires that we provide parsers which sugar this syntax and allow more friendly expressions for many of the items. To this end, we will allow '[]'(<T>,<L>) to be written as [<T> | <L>]. In fact, if <T> and <U> are any terms, we will allow the user to write

'[]'(<T>,<U>) as [<T> | <U>].

The list

'[]'(<T1>,'[]'(<T2>,'[]'))

can be written as

[<T1>, <T2>], etc.

Definition 4.7. If <A> is any literal and <c>,<c1>,...,<cn> are any constants, then

all(':'([<c1>,...,<cn>], <A>)) (4.8)

and

all(':'(<c>, <A>)) (4.9)

are both *universal assertions*. If <A> contains no logical variables, they are both called *facts*. All facts are *clauses*. The literal <A> is also called a *fact matrix*.

The sugared syntax allows (4.8) and (4.9) to be written respectively in the forms:

all [<c1>,...,<cn>] : <A>. (4.10)

all <c> : <A>. (4.11)

Definition 4.12. The class of *goals* is defined recursively as follows:

(.1) The distinguished term 'true' (which is a literal) is a goal.

(.2) If <G> is a goal and <L> is a literal, then '&'(<L>,<G>) is a goal. For readability, goal expressions such as

'&'(<L1>,<L2>)

can be written as

<L1>& <L2>,

with '&' associating to the right.

Definition 4.13. An *implication symbol* is one of the symbols '<—', ':-', or 'if'.

Definition 4.14. If <A> is any literal, if <B> is any goal, if <c>,<c1>,...,<cn> are any constants, and if <I> is an implication symbol, then

all(:([<c1>,...,<cn>], <I>(<A>, <B>)) (4.15)

and

all(:(<c>, <I>(<A>, <B>)) (4.16)

are both called *universal implications*. If neither <A> nor <B> contains any logical variables, both (4.15) and (4.16) are called *rules*. Any rule is a *clause*. The expression <I>(<A>,<B>) is called a *rule matrix*.

For readability, (4.15) and (4.16), say with <I> being '<—', are written respectively as:

all [<c1>,...,<cn>] : <A> <— <B>. (4.17)

all <c> : <A> <— <B>. (4.18)

Definition 4.19. The class of *theories* is defined recursively as follows:

(.1) The constant ' empty_theory ' (which is a literal) is a theory.

(.2) If <C> is any clause and <T> is any theory, then

&(<C>, <T>) (4.20)

is a theory. For readability, (4.20) can be written as

<C> & <T>, (4.21)

with '&' associating to the right.

Definition 4.22. If <B> is any term and <c1>,...,<cn> are any constants, then

:([<c1>,...,<cn>], <B>) (4.23)

is a *special form*. Note that n=0 is allowed so that

:([], <B>) (4.24)

is a special form. For readability, (4.23) and (4.24) are written respectively as:

[<c1>,...,<cn>] : <B> (4.25)

[] : <B> (4.26)

5. Programming Examples: Poirot

One of the immediate uses to which one can put theories is the handling of varying points of view, for example as considered by Fain et al. in Section 3 of [ROSIE]. To quote that paper:

> The "problem statement" in this domain is straightforward: given some set of facts and some set of participants, the detective, Poirot, must uncover the information necessary to deduce which participants might be guilty. Poirot uncovers information by mediating a dialogue between the user and each participant. Poirot then uses the information gleaned from the interrogation to make his deductions. In other words, the user asks the questions and Poirot makes the inferences. Unfortunately, each potential suspect has his or her unique viewpoint of and knowledge about the situation. Thus, in terms of implementation, we need some way of simulating the privacy of each participants' memory and some mechanism or mechanisms for simulating the question/answer protocol of interrogation.

The mechanism we use here is that of theories: each participant is represented by a separate theory. Thus, for example, the theory representing Poirot is named poirot and consists of the following clauses:

```
rich(mary).
involved(sara).
involved(john).
```

The theory which represents John is named john and consists of the clauses

```
need(john,money).
married_to(john,mary).
loved(john,mary).
```

while the theory representing Sara is similarly named sara and consists of the clauses

```
sister(sara,mary).
loves(sara,john).
did_not_love(john,mary).
loves(john,sara).
```

The theory common contains knowledge regarded as common to all participants:

```
man(john).
man(poirot).
woman(mary).
woman(sara).
found_dead(mary).
detective(poirot).

all individual :
    person(individual)<—
        man(individual).
all individual :
    person(individual)<—
        woman(individual).
all [individual, person] :
    married(individual, person) <—
        married_to(person, individual).
all [individual, person] :
    married(individual, person)<—
        married_to(individual, person).
all [individual, person] :
    married(individual) <—
        wife(person, individual).
all [individual, person] :
    married(individual) <—
        husband(person, individual).
all [individual, person] :
    married(individual) <—
        spouse(person, individual).
all [firstPerson, secondPerson] :
    related(firstPerson, secondPerson) <—
        married_to(firstPerson, secondPerson).
all [firstPerson, secondPerson] :
    related(firstPerson, secondPerson) <—
```

```
            married_to(secondPerson, firstPerson).
        ...
    all [firstPerson, secondPerson] :
        related(firstPerson, secondPerson) <—
            sister(firstPerson, secondPerson).
        ...
    all [firstPerson, secondPerson] :
        related(firstPerson, secondPerson) <—
            daughter(firstPerson, secondPerson).
```

(Mary, being deceased, has no theory representing her interests.) Recall that Poirot listens in on the interrogation that we conduct with John and Sara, and then makes his deductions from the evidence accumulated (i.e., the positive responses that John or Sara makes.) Poirot thus requires a theory which represents his "theory of evidence": the rules whereby he can conclude that someone is a suspect and what their possible motive might be. This is represented by the theory named suspct:

```
all [person, otherPerson, victim] :
    suspect(person, jealousy) <—
        loves(person, otherPerson) &
        married(otherPerson, victim) &
        found_dead(victim).

all [person, victim] :
    suspect(person, greed) <—
        need(person, money) &
        found_dead(victim) &
        rich(victim) &
        related(person, victim).

all [person, otherPerson, victim] :
    suspect(person, revenge) <—
        loved(otherPerson, victim) &
        not(otherPerson = person) &
        found_dead(victim) &
        rejected_by(person, otherPerson).

all [person, otherPerson] :
    rejected_by(person, otherPerson) <—
```

```
        loves(person, otherPerson) &
        not(person = otherPerson) &
        not(loves(OtherPerson, Person)).
```

The workhorse part of the program is contained in the theory labelled detect. The entry to the entire program is the zero argument predicate detectiveStory. The first step of this predicate is to assemble the set (list) of suspects according to Poirot -- this is called Candidates. The next step is to operate on Candidates using the predicate doInterrogations, producing a list called Suspects.

It is during the running of doInterrogations that the user asks questions of the candidates and Poirot "listens." The information Poirot finds interesting is recorded in the list Suspects. Specifically, the questions which each candidate answered positively are recorded and associated with the candidate in the list Suspects. Next, Poirot reorganizes the evidence using the predicate assembleEvidence. He then uses the individually recorded evidence (Suspects) together with the reorganized evidence (TotalEvidence) in conjunction with his theory "suspct" to draw conclusions about the individuals. He reports these conclusions to the user via the predicate reportOn.

```
all [person, candidates, suspects, totalEvidence] :
    detectiveStory <—
        demo(poirot, setOf(person, involved(person), candidates)) &
        doInterrogations(candidates, suspects) &
        assembleEvidence(suspects, totalEvidence) &
        reportOn(suspects, totalEvidence).
 doInterrogations([],[]).

all [person, rest_of_candidates, reasons,rest_of_suspects] :
    doInterrogations([person | rest_of_candidates],
            [suspect(person,reasons) | rest_of_Suspects]) <—
        interrogate(person, reasons) &
        doInterrogations(rest_of_Candidates, rest_of_Suspects).
```

The predicate doInterrogations simply recurs down the list of Candidates, interrogating each person and recording the Reasons for which that person might be a suspect.

```
all [person, reasons] :
    interrogate(person, reasons) <—
```

```
        conductInterrogation(person, [], reasons).


all [person, currentReasons, finalReason, instructions] :
    conductInterrogation(person, currentReasons, finalReasons) <—
        obtainFromUser(instructions) &
        actOn(instructions, person, currentReasons,  finalReasons).
```

The predicates conductInterrogations and interrogate are mutually recursive. The positive answers to questions are accumulated in the second argument of conductInterrogations. When the user's instruction is to quit this particular interrogation, actOn causes the accumulated answers to be returned by conductInterrogations in its third argument. Note that the privacy of individual views is achieved in actOn by deducing the user's Query from the theory "common" combined with the theory defining the individual suspect (e.g., "john").

```
all command :
    obtainFromUser(command) <—
        nl & write('>') & read(command).

all [person, currentReasons] :
    actOn(done, person, currentReasons, currentReasons).

all [person, currentReasons, finalReasons] :
    actOn(interrogate, Person, CurrentReasons, FinalReasons) <—
        write('Interrogating ') & write(person) & nl &
        conductInterrogation(person, currentReasons, finalReasons).

all [query, person, currentReasons, finalReasons, newReasons]:
    actOn(query, person, currentReasons, finalReasons) <—
        demo(common  &  person, query)  &
        respond('Yes, ', query) & nl &
        addTo(CurrentReasons,Query,NewReasons)  &
        conductInterrogation(person, newReasons,  finalReasons).

all [query, person, currentReasons, finalReasons] :
    actOn(query, person, currentReasons, finalReasons) <—
        respond('No, it is not the case that ', query) & nl  &
        conductInterrogation(person, currentReasons, finalReasons).
```

This completes the definition of doInterrogations. The predicate assembleEvidence simply forms a virtual union of the reasons associated with each suspect:

```
assembleEvidence([], empty_theory).

all [person, reasons, rest_susp, other_ev] :
    assembleEvidence([suspect(person,reasons) | rest_susp],
            other_ev & reasons) <—
        assembleEvidence(rest_susp, other_ev).
```

The final top-level predicate is reportOn, which handles both carrying out Poirot's final deductions regarding the status of each suspect together with reporting on the evidence and the deductions to the user.

```
reportOn([],_).

all [person, reasons, rest_susp, totalEvidence] :
    reportOn([suspect(person, reasons) | rest_susp], totalEvidence) <—
        write('Evidence concerning ') &
        write(person) & write(':') & nl &
        exhibit(reasons) &
        determineSuspectStatus(person, reasons, totalEvidence) &
        reportOn(rest_susp, totalEvidence).
```

Poirot's attempts to deduce which persons are really suspects is carried out in the predicate determineSuspectStatus by the call to demo. Note that the theory under which the attempted deduction takes place consists of the theory representing Poirot combined with his theory of suspects, the common knowledge, and the TotalEvidence acquired during the interrogations. The full three-argument version of demo is used so as to obtain the actual proof which is then used in presenting the conclusions to the user in discussSuspectStatus.

```
all [person, reasons, totalEvidence, motive, deduction] :
    determineSuspectStatus(person, reasons, totalEvidence) <—
        demo(suspct & common & poirot & totalEvidence,
        suspect(person,motive),proof(deduction)) &
        discussSuspectStatus(person,motive,reasons,totalEvidence,
                                                deduction).
```

```
all person : determineSuspectStatus(person, _, _) <—
    discussSuspectStatus(person, _, _, _, innocent).
```

The remaining predicates are concerned with reporting to the user. We will omit their details. Below is a transcript of part of one run of the program (User input is shown in **boldface**.)

>interrogate.
Interrogating john
>need(john,money).
Yes, john need money
>related(john,mary).
Yes, john related mary
>loved(john, mary).
Yes, john loved mary.
>loves(john, sara).
No, it is not the case that john loves sara.
>loves(sara, john).
No, it is not the case that sara loves john.
>done.
>interrogate.
Interrogating sara.
>need(sara,money).
No, it is not the case that sara need money
>related(sara,mary).
Yes, sara related mary
>loved(john, mary).
No, it is not the case that john loved sara.
>loves(john, sara).
Yes, john loves sara.
>loves(sara, john).
Yes, sara loves john.
>done.

Evidence concerning john:
    john need money
    john related mary
    john loved mary

john is a suspect. Motive: greed.

Poirot concludes that john is a suspect with the motive of greed. Here's his reasoning:

john related mary is established.
mary is rich is established.
mary is found_dead is established.
john need money is established.
john suspect greed because of the rule
    john suspect greed holds if
        john need money &
        mary is found_dead &
        mary is rich &
        john related mary.

Evidence concerning sara:
    sara related mary
    john loves sara
    sara loves john

sara is a suspect. Motive: jealousy.

Poirot concludes that sara is a suspect with the motive of jealousy. Here's his reasoning:

mary is found_dead is established.
john married mary is established.
sara loves john is established.
sara suspect jealousy because of the rule
    sara suspect jealousy holds if
        sara loves john &
        john married mary &
        mary is found_dead.

Notice the two different points of view expressed by the different answers john and sara give to the questions presented.

6. Bottom-Up Parsing

Our next programming example is an unusual construction of a bottom-up parser using the ability to generate new grammars (represented by theories) on the fly. The grammars are expressed with the rules of Definite Clause Grammars (cf. Pereira and Warren [1980]). They process a list of Prolog terms representing the tokens of the sentence to be parsed. At all points in the algorithm, the processing is relative to a current extended grammar XG. (The original grammar G is always passed along for use in generating the next extended grammar.) The essence of the algorithm is as follows:

1. If the sentence is the empty list and the rule
   (sentence -->)
   belongs to XG, terminate successfully.

2. If the sentence is non-empty and the rule
   (sentence -->)
   belongs to XG, backtrack.

3. Otherwise, let T be the left-most token, let R be the remainder of the sentence list, and proceed as follows:

   a) Select a "grammar fact" from G which will reduce the selected token to a non-terminal grammar symbol and apply it to T, yielding T1.

   b) Select a grammar rule from XG with a non-terminal symbol NT as its head such that the first symbol in the body of the rule matches T1; let RB be the remainder of the body of this rule.

   c) Use XG, NT, and the reduced rule
      NT --> RB
   to construct a new extended grammar XXG; replace XG by XXG and goto step 1.

Step 3c) expands to the following algorithm:

i) Collect the set K of all reduced rules of the form
   H --> B,

where the rule

H --> NT, B

belongs to XG; note that B may be empty.

ii) If K contains a rule

(H1 -->),

choose one such, and let XXG be the set of all rules of the form

H2 --> B2,

where the original grammar G contains the rule

H2 --> H1, B2.

iii) Otherwise, let XXG be K together with the rule

NT --> RB.

The code for implementing a simple version of this in metaProlog runs as follows:

```
all [G, S] : parse(G, S) <— pp(G, G, S).

all [G, XG] : pp(G, XG, []) <— belongs((sentence -->), XG) & fail.

all [G, XG, T, R, T1, NT, RB, XXG] :
    pp(G, XG, [T | R]) <—
        belongs((T1 --> [T]), G) &
        belongs((NT --> T1, RB), XG) &
        reduce(G, XG, NT, RB, XXG) &
        pp(G, XXG, R).

all [G, XG, NT, RB, XXG, H1] :
    reduce(G, XG, NT, RB, XXG) <—
        belongs((H1 --> NT), XG) &
        filter(G, H1, XXG).

all [G, XG, NT, RB, XXG, H1, G1] :
    reduce(G, XG, NT, RB, XXG) <—
        not(belongs((H1 --> NT), XG)) &
        filter(XG, NT, G1) &
        addTo(G1, (NT --> RB), XXG).
```

The predicate filter(T1, X, T2) is defined to hold if T1 is a theory and T2 is the theory consisting of all those rules of the form

H --> B

where

H--> X, B

belongs to T1. Something like this filter, but more general, is a candidate for being a built-in for metaProlog, but more experimentation is necessary before a decision is made. [In general, there is considerable room for discovery and specification of theory-manipulation predicates.] It is possible to simulate filter(T1, X, T2) by using setOf to create the list of all axioms of T1, and recusrsively process that list to build up T2 from the empty theory.

7. Co-routining and Parallelism

As part of our program of providing powerful tools for AI programming, we seek to offer the programmer control of stream-based communication between concurrent processes, while still holding to our program of preserving the essential elements of Prolog semantics. In the logic programming context, this amounts to implementing some form of and-parallelism. The most straight-forward sort of and-parallelism to attack is simple producer - consumer computations. However, since the implementation of producer - consumer relations in which the producer is allowed to non-determinately reconsider the stream it has produced is difficult to say the least, we restrict ourselves to determinate and-parallel situations. Other approaches to parallelism in Prolog (e.g., Parlog (Clark and Gregory [198?]) or Concurrent Prolog (Shapiro [1983])) achieve this restriction by introducing committed choice. However, while preserving the correctness of the computations, this approach loses Prolog's deductive completeness. In contrast, we preserve both the correctness and completeness by restricting ourselves to running in parallel only producer - consumer computations in which the production of the stream is determinate. (Note that the computation of the elements of the stream may involve non-determinate aspects; it is simply at the point of adding a new element to the stream that the producer must act determinately. Also, consumption of the stream may be entirely non-determinate.) The essential point appears to us that it is not really the processes which must be forced to be determinate, but rather the communication between them. Thus our approach is to force the producing process to determinately fill the communication buffer; all else can be non-determinate.

We have identified two useful classes of producer - consumer computations which meet our requirement (and the possibility of others certainly exists).

The first is the (lazy) production of sets via complete exploration of a search tree (i.e., the lazy form of Prolog's setof construct) and the production of streams by determinate tail-recursive procedures. These are indicated in metaProlog programs by the constructs

all_solutions(Template, Goal, Stream)

and

streamOf(Goal, Stream).

We see these as entirely encapsulated independent computations: their only method of communication with parent or sibling processes is via the stream variable. Every element of the stream must be ground. If the producing process would have otherwise produced a partially instantiated term as a stream element, that term must be converted to a ground term by use of the 'naming' or 'indicating' operator discussed above in conjunction with quantification. The same restrictions clearly must apply to the Goal argument of both stream_of and all_solutions. One method of implementation is that of producer variables. The first invocation of Goal binds the variable Stream to a buffer together with a description of Goal and its environment. Subsequent attempts to access the variable stream by the consumer causes Goal to be run through one cycle of its computation, binding Stream to a cons cell whose first element is the item produced and whose second element is a description of the rest of the buffer together with the current state of the computation of Goal. It is important to recognize that the producer variable does not act like normal Prolog variable. Indeed, since any attempt to match a non-variable term against an element of the stream causes the stream element to be instantiated to a ground term by the producer, and since the producer is determinately committed to the binding it produces, producer variables behave for all intents and purposes as ground objects. Thus it is perfectly permissable for producer variables to appear in the Goal arguments of other producer processes. This allows for two-way communication between producers. Process synchronization is achieved by requests for bindings passed from process to process. It is clear that the two communicating processes must created simultaneously. The construct

simultaneous(Process1, Process2)

achieves this effect. It can be invoked with any number of arguments.

Because we see these processes as entirely sealed computations with their own environments, it is possible, in appropriate hardware settings, to run them truly in parallel, allowing the producing process to fill the buffer

up to some pre-set limit or even run to completion when the stream is finite. On sequential hardware, the implementation is simple co-routining of the producer and consumer, with the additional overhead entirely localized in the communication -- there is no slow down of the basic Prolog computation. In particular, the computational children of the Goal of one of these processes do not inherit the parallel mode: they run as normal Prolog processes. It should be possible to mix parallel and co-routined execution with no change to the program or its behavior. Finally, while we have not attempted to do so, it seems evident that or-parallelism could be introduced with a stream operator whose top level was expanded in an or-parallel manner. One might even introduce committed-choice versions of such an operator without disturbing the semantics of the rest of the system.

8. Programming Examples: The Inland Spills Expert

Here we discuss an adaptation of a program to manage inland chemical spills at the Oak Ridge National Laboratory. The problem is discussed in detail in Hayes-Roth, et al.[1984]. Our metaProlog program for this problem was strongly influenced by the Rosie SPILLS program (Fain et al.[1982]), from which the following problem statement was drawn:

> The SPILLS program locates and identifies hazardous chemical spills, given a database describing the location of the spill, the location of chemical storage containers, and a description of the drainage network. SPILLS evolved as an answer to a problem posed at the expert Systems Workshop in San Diego, August 1980 ...The problem involved the creation of an on-line assistant to aid a crisis control team in the location and containment of chemical spills at the Oak Ridge National Laboratory. Two experts in the field plus a preliminary report ... provided the necessary expertise.
>
> The Oak Ridge Laboratory has approximately 200 buildings scattered over a 200-square-mile area, many storing hazardous chemicals in containers ranging in size from small 1-gallon bottles or cans to huge 5,000-gallon storage tanks. The drainage network under the building collects all spills and discharges them into White Oak Creek, a waterway running along one side of the lab's complex. When chemical discharges are noticed in the creek they must be traced back to some source (a storage container in or near a laboratory building) so the leak can be stopped and the spill contained.
>
> The SPILLS program attempts to locate the source of the spill by tracing the flow of spilled material through the drainage basin back to the source. This search method requires a human assistant who must go out in the field and actually look into the drainage networks at various check points (usually manholes) to see if the spill material is there. There are so many manholes (hundreds) that it is not practical to check them all for traces of the spill. Instead, the program uses the information at its disposal to decide which

checkpoint would provide the maximum amount of information at any given time, and recommends that the assistant examine that checkpoint. After the program is told the result of that examination, it recalculates the new optimal checkpoint, and the process continues until the source is found.

Besides processing reports on the location of the spill material, the program processes reports describing the characteristics of the spill material. It attempts to determine what the material is and how much of it has been spilled. This information, in turn, helps reduce the number of possible locations to be checked.

One of the many drainage basins at Oak Ridge is shown in Figure B.8.1 below.

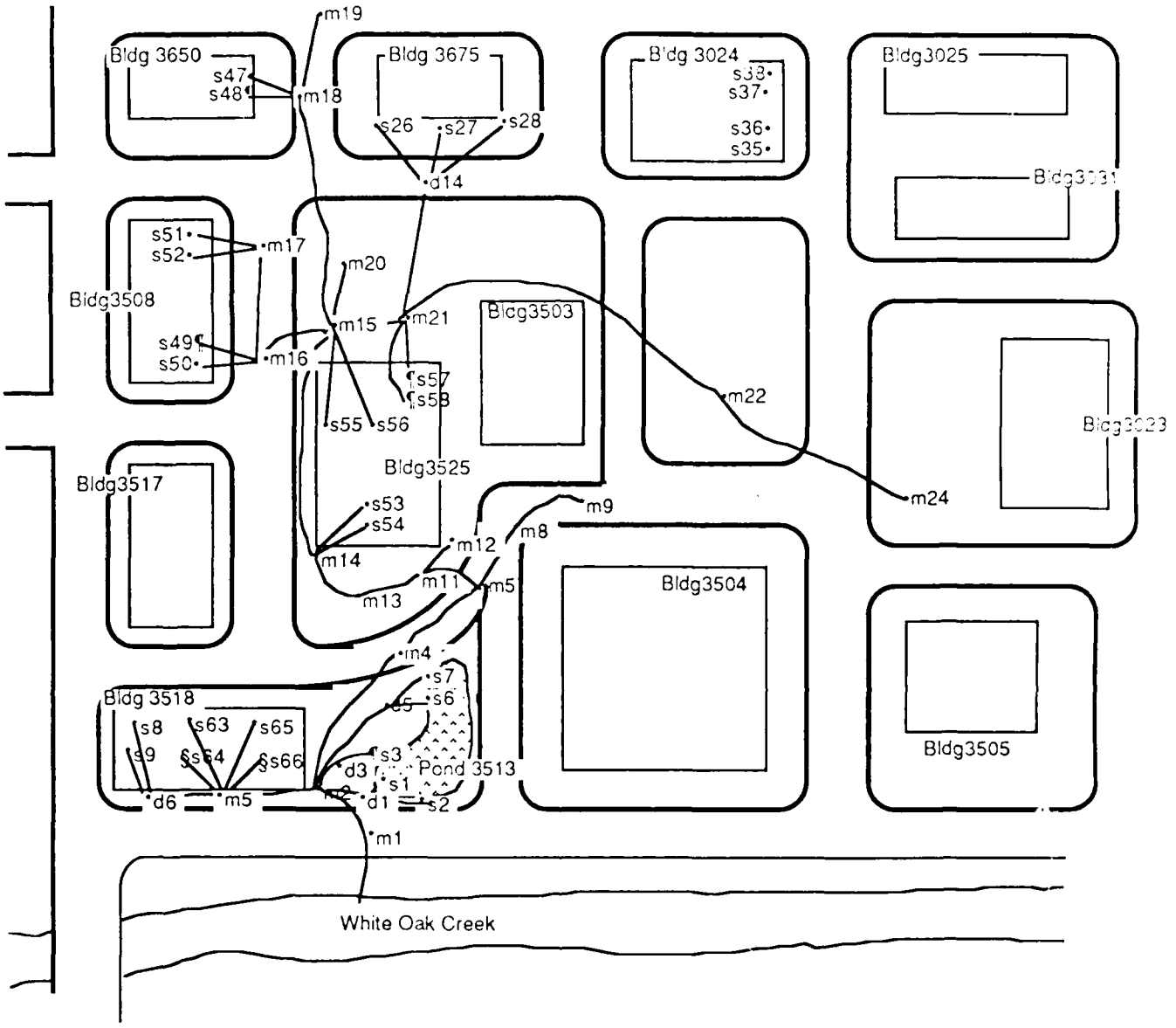


Figure B.8.1.
Part of One of the Drainage Basins at Oak Ridge National Laboratory.

It is evident that the basin forms a tree with its root at the White Oak Creek effluence and its tips consist of the various sources. The problem then is one of exploring this search tree starting at the root (where the spill is first observed) and locating the offending tip (a leaky tank). The difficulty in this exploration which involves a human assistant going out and looking into the manholes) is the large size of the tree. Thus the program, just as a human, will attempt to apply knowledge to minimize

the portion of the tree which must be examined.

Conceptually, at the outset of the search all of the tips are possible sources of the spill. As knowledge (of the nature of the material and of the manholes at which it has been observed) is accumulated, various of the tips are eliminated as possible sources. Our metaProlog program, named OakRidge, emulates this approach by maintaining a dynamic theory representing the current state of its knowledge. At the outset, it consists solely of a collection of assertions to the effect that each of the sources in the basin is a possible source of the spill:

```
possibleSource(s(1)).
possibleSource(s(2))
. . . .
possibleSource(s(70)).
```

As the search progresses and various sources are eliminated, the appropriate possibleSource(s(N)) assertions are deleted, while assertions regarding the nature and properties of the spill material are added along with assertions about the manholes at which it has been observed.

The knowledge which the program possesses at the outset is broken up into various static theories which are utilized by the reasoning processes. The knowledge of the topology of the network, the nature of its nodes, and the nature of each of the sources is contained in a theory called srcs:

```
isPond(pond(3513)).
outfall(woc(6)).
isBuilding(building(3023))
. . .
isBuilding(building(3550)).

all N : isDrain(drain(N)) <— between(0, N, 16).
all N : isManhole(m(N)) <— between(0, N, 47).
all N : isSource(s(N)) <— between(0, N, 71).
all N : near(s(N), pond(3513)) <— between(0, N, 8).
all N : in(building(3023), s(N)) <— between(43, N, 46)
. . .
all N : in(building(3504), s(N)) <— between(10, N, 17).
all N : in(building(3504), s(N)) <— between(67, N, 70)
```

```
. . .
parent(woc(6), m(1)).
parent(m(1), m(2)).
parent(m(2), m(3)).
parent(m(2), m(4)).
. . .
all N : parent(m(5), drain(N)) <— between(7, N, 10).
. . .
all [M, N] : parent(drain(N), s(M)) <— between(2, N, 5) & M is N+1.
. . .
contains(s(1), gallons(2000), [transformer, oil]).
contains(s(2), gallons(1000), [gasoline]).
contains(s(3), gallons(10), [acetic, acid]).
. . .
```

Other static theories contain knowledge about how to infer the nature of the spill material from its properties, how to infer the next manhole to examine, and how to eliminate possible sources. These will be described below. The top level of the program appears as follows:

```
oakRidge <— investigate(null).

all [currentData, updatedData, d0, d1, d2, d3] :
    investigate(currentData) <—
        write('Report please:') & nl &
        getReport(currentData,updatedData) &
        workOnMaterialType(updatedData,d0) &
        workOnMaterial(d0,d1) &
        workOnMaterialVolume(d1,d2) &
        workOnMaterialSource(d2,d3) & ! &
        dispatch_investigate(d3).

all updatedData :
    dispatch_investigate(updatedData) <—
        finished(updatedData) & !.

all data : dispatch_investigate(data) <—
    ! & investigate(data).
```

The predicate getReport is the "natural language" front-end which

obtains information to the user. The details of its definition are included in the appendix to this section. The variable CurrentData is bound to a theory which represents the current information regarding the spill being investigated. This is extended by getReport with the information obtained. The workOn_____ predicates are concerned with inferring the general nature, specific identitiy, volume of, and source of, the spill. They each take the current information as input, and add to it any inferences they may make to produce their output. Finally, dispatch_investigate determines whether or not the source has been determined, and hence, whether or not the investigation should be (tail recursively) continued.

Consider the predicate workOnMaterialType. It is concerned with the problem of inferring the general nature of the spill material. It bases its deductions on the current information regarding the spill material (the variable UpdatedData) and a small theory "typeOfMaterial" which encodes the "expertise" for inferring the types of spill materials in this setting. The definition of workOnMaterialType runs as follows:

```
all [data, extendedData, x ] :
    workOnMaterialType(data, extendedData) <--
        write('Trying to determine material type...') & nl &
        demo(typeOfMaterial & Data,type_of_material(spill,x)) &
        ! &
        addTo(data, type_of_material(spill,x), extendedData) &
        write('The type of the spill material is ') &
        print(x) & nl & nl.

all data :
    workOnMaterialType(data, data) <--
        write('Can''t determine the type of the spill material now...') &
        nl & nl.
```

The theory typeOfMaterial contains the following axioms:

```
all n :  type_of_material(spill, [oil]) <--
    appears(spill_solubility, [low]) &
    approximates(ph_of_spill, [n]) & 5 < n & n < 9.

all n :  type_of_material(spill,[base]) <--
    appears(spill_solubility,[high]) &
    approximates(ph_of_spill,[n]) & 8 < n.
```

```
all n : type_of_material(spill,[acid]) <—
    appears(spill_solubility,[high])  &
    approximates(ph_of_spill,[n])  & n < 6.
```

The predicate workOnMaterial attempts to infer the specific composition of the spill. It bases its work on the current information at the time of its invocation (the theory d0) together with the theory "materialType" contains the expertise for inferring the spill composition. The definition of workOnMaterial runs as follows:

```
all [data, extendedData, x] :
    workOnMaterial(data, extendedData) <—
        write('Trying to determine material...') & nl  &
        demo(materialType & data, consists(spill,of(x    & ! &
        addTo(data, consists(spill,of(x)), extendedData)  &
        write('The spill consists of ')  & write(x) & nl & nl.

all data : workOnMaterial(data,data) <—
    write('Can"t determine what the material is now...')    & nl & nl.
```

The theory materialType contains the following axioms:

```
consists(spill,of([sulpheric,acid])) <—  iss(sulphate_ion_test,[positive]).

consists(spill,of([gasoline])) <—
    type_of_material(spill,[oil]) & smells(spill,of([gasoline])).

consists(spill,of([diesel,oil])) <—
    type_of_material(spill,[oil]) & smells(spill,of([diesel,oil])).

consists(spill,of([acetic,acid])) <—
    type_of_material(spill,[acid]) & smells(spill,of([vinegar])).

consists(spill,of[hydrochloric,acid])) <—
    type_of_material(spill,[acid]) &
    has(spill,[pungent,'/',choking,odor]).
```

While the content of the rules in these theories is not particularly deep, nonetheless, they exhibit the necessary characteristic of expert

system rules: the clear expression of whatever expertise they embody.

The code for trying to determine the source of the spill runs as follows:

```
all [data, extendedData, xData1, xData2] :
    workOnMaterialSource(data, extendedData) <—
        write('Trying to determine source...') & nl &
        updateDetected(data, xData1) &
        eliminatePossibleSources(xData1, xData2) &
        checkForSource(xData2, extendedData).

all data :
    workOnMaterial(data,data) <—
        write('Can''t determine what the material is now...')
        & nl & nl.

all [dataIn, dataOut, prevHighNode, newHighNode]:
    updateDetected(dataIn, dataOut) <—
        axiomOf(dataIn, highestNodeDetected(prevHighNode)) &
        demo(dataIn, detected(spill, at(newHighNode))) &
        demo(srcs, above(newHighNode, prevHighNode)) & ! &
        dropFrom(dataIn, highestNodeDetected(prevHighNode), data1) &
        addTo(data1, highestNodeDetected(newHighNode), dataOut).

all data:
    updateDetected(data, data).

all [dataIn, dataOut, conseq, fact,
                        possibleConditions, excludedSrcs, xData]:
    eliminatePossibleSources(dataIn, dataOut)<—
        setOf(conseq,
                ( axiomOf(dataIn, fact) &
                  demo(seekSrc, conseq, bottom_up(fact)),
                 possibleConditions) &
        checkOut(srcs & dataIn, possibleConditions, [], excludedSrcs) &
        revise(dataIn, excludedSrcs, xData) &
        checkForBypass(xData, dataOut).

all [theory, accumulated, final]:
    checkOut(theory, [], accumulated, final)<—
        final = accumulated.
```

```
all [theory, restConds, accumulated, final]:
    checkOut(theory, [true | restConds], accumulated, final)<--
        checkOut(theory, restConds, accumulated, final).

all [theory, S, C, restConds, accumulated,
                    implications, newAccumulated, final]:
    checkOut(theory, [true | restConds], accumulated, final)<--
        setOf(impSrc(S), demo(theory, (C & source(S) )), implications) &
        append(accumulated, implications, newAccumulated) &
        checkOut(theory, restConds, newAccumulated, final).
```

The theory seekSrc contains rules such as the following:

```
all [S, material, N, otherMaterial]:
    impossibleSource(S) <--
        consists(spill, of( material) ) &
        contains(S, gallons(N), otherMaterial) &
        not( material = otherMaterial).

all [S, V, SomeMaterial]:
    impossibleSource(S)<--
        volumeOf(spill, gallons(V)) &
        contains(S, gallons(N), SomeMaterial) &
        N < V.

    .....
```

9. Fault Detection in Digital Circuits.

In this section we describe approaches to fault-detection in digital ciruits based on the ideas of Esghi [], extended to a hierarchical setting similar to that of Genesereth[].

9.1. Circuit Description and Simulation

For the purposes of fault-finding, the devices must be described in some sort of predicate calculus formalism. The exact format is unimportant. For the purposes of the simple example we consider, we label the gates and lines (nodes) of a combinational circuit as indicated in Figure B.9.1.

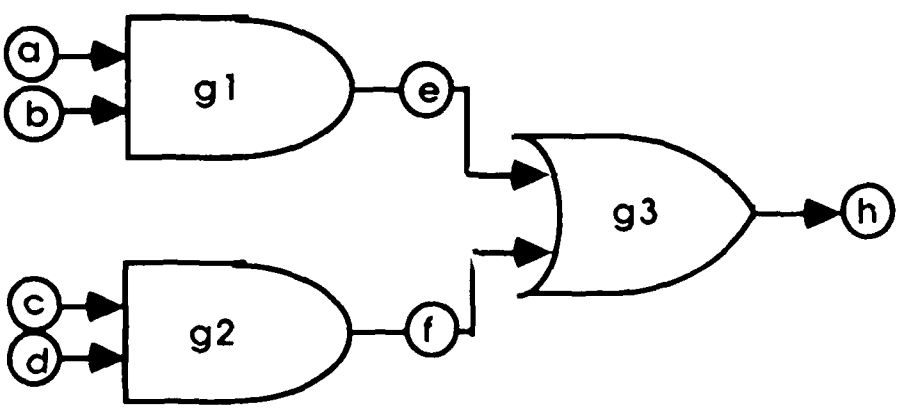


Figure B.9.1. A Simple Circuit.

The predicate

andGate(G, In1, In2, Out)

expresses that G is an and-gate with input lines In1 and In2, and output line Out. Similarly for orGate. The topological description of the circuit is contain in the theory c1:

```
andGate(g1, a, b, e).
andGate(g2, c, d, f).
orGate(g3, e, f, h).                                    (9.1)
inputNodes([a, b, c, d]).
outputNodes([h]).
```

The predicate inputNodes holds of the list of inputs to the circuit as a whole, while the predicate outputNodes holds of the list of output nodes for the entire circuit. The behaviors of the circuit components are described in the theory tt (for truth tables):

```
all [Gate, In1, In2, Out] :
andTable(Gate, In1, In2, Out) <—
  not(exceptional(Gate)) &
  standardAnd(In1, In2, Out).

all [Gate, In1, In2, Out] :
orTable(Gate, In1, In2, Out) <—                         (9.2)
  not(exceptional(Gate)) &
  standardOr(In1, In2, Out).

standardAnd(high, high, high).

all In2 : standardAnd(low, In2, low).
all In1 : standardAnd(In1, low, low).
all In2 : standardOr(high, In2, high).
all In1 : standardOr(In1, high, high).

standardOr(low, low, low).

user choice(delTab).
```

The significance of the predicates exceptional and user choice will be described later. The topology and component behaviors can be used to predict the circuit outputs given the inputs as described in the theory laws:

```
all InputList
predict(InputList, [])
    write('predict ')
```

```
all [InputList, Node, Rest_Output_Nodes, State,    Rest_Output] :
predict(InputList, [Node | Rest_Output_Nodes],
                    [out(Node,State) | Rest_Output]) <--                    (93)
     state(Node, InputList, State) &
     predict(InputList, Rest_Output_Nodes,
                    Rest_Output_Nodes, Rest_Output).

all [Node, InputList, NodeState, GateName, InputLine1, InputLine2] :
state(Node, InputList, NodeState) <--
     andGate(GateName, InputLine1, InputLine2, Node)  &
     state(InputLine1, InputList, Line_1_State)  &
     state(InputLine2, InputList, Line_2_State)  &
     andTable(GateName, Line_1_State, Line_2_State, NodeState).

all [Node, InputList, NodeState, GateName, InputLine1, InputLine2] :
state(Node, InputList, NodeState) <--
     orGate(GateName, InputLine1, InputLine2, Node)  &
     state(InputLine1, InputList, Line_1_State)  &
     state(InputLine2, InputList, Line_2_State)  &
     orTable(GateName, Line_1_State, Line_2_State, NodeState).

all [Node, NodeState, RestInput] :
state(Node, [in(Node, NodeState) | Rest_Input], NodeState).

all [Node, Rest_Input, NodeState] :
state(Node, [_ | Rest_Input], NodeState) <--
     state(Node, Rest_Input, NodeState).
```

The predicate predict can be used to simulate the action of circuits. Thus, for example, to simulate the action of the sample circuit described above when the input lines a, b, c, and d are respectively set to high, low, low, and low, one would run the metaProlog goal

```
demo(c1 & tt & laws,                                                    94
          predict([in(a,high),in(b,low),in(c,low),in(d,low)], Out))
```

which would be solved yielding the value Out = [out(h,low)

9.2. Fault Diagnosis

This use of predict for simulation of circuit behavior is interesting, but not very exciting. However, the predicate predict can be used in metaProlog to organize a very promising approach to fault diagnosis in digital circuits. This approach relies heavily on the ability to manipulate and create theories. The essence of the approach is this. We are given a description D of the circuit under diagnosis, together with an input-output pair If, Of for the actual circuit in which the output behavior Of is faulty, that is not what predict would calculate based on D. The heart of the approach is to infer a new description Dt by a minimal perturbation of D such that using Dt, predict would correctly calculate the pair If, Of. That is, Dt will be a correct description of the faulty circuit. By comparison of D and Dt, the fault can then be located. We will assume that the original circuit D is reachable and observable, and moreover, that the faulty behavior is due to a single pin of a single gate being stuck either high or low. However, both assumptions can be weakened. In the absence of observability, the output of the algorithm will be a list of candidate descriptions of the faulty circuit. Multiple faults or short circuits can be attacked by modifying the hypothesis generation stage of the algorithm.

The basic diagnostic algorithm proceeds as follows. 65

1: From D and If, Of construct a set HYP = {H1, H2, ...} of theories such that the following conditions hold:

1.1) For all i, from Hi and If predict If, Of holds.
1.2) For some i, Hi is a minimal deviation from the faulty circuit.

2: If cardinality(HYP) > 1 then reject HYP.

3:
3a) [illegible] HYP
3b) [illegible]
3c) [illegible]

4: [illegible]

5: Delete from HYP all Hi for which the goal

demo(Hi & laws, predict(Id, Od))

fails.

6: Goto step 2.

Steps 2 - 6 of this algorithm constitute a reasonably standard "test and eliminate" loop which we will discuss later. The most interesting part of the algorithm is its first step, that of generating the candidate descriptions of the faulty circuit. (Note that the entire algorithm is a classic "generate and test.") What is needed is a heuristic to guide this generation through the combinatorial nightmare of all possible circuit descriptions. The key is provided by the mathematician's observation that even failed attempts at proofs are often useful in guiding a search for a correct proof (as strikingly illustrated by Kemp's false proof of the four-color conjecture and the Appel-Haken correct proof.) First note that since (If, Of) is a faulty I-O pair for the correct circuit D, the goal

demo(D & laws, predict(If, Of)) (9.6)

must fail. However, in the process of failing this goal, the metaProlog system systematically explores the search tree for this goal. Each of the branches of this tree is a failed proof of the goal.

What we propose to do is what a mathematician normally does not permit himself, namely to ask the question: Can I modify the axioms of the theory D to make this failed proof into a correct proof relative to the modified theory? In our case, we will only allow modifications to D which reflect 'sticking' a pin of a gate at either high or low. Thus we will generate all those modifications of D which

1. are obtained by 'sticking' one pin of one gate at high or low,

and

2. allow the goal (9.6) to be successful.

Under our basic assumptions, this procedure will satisfy the requirements of step 1 of the algorithm, and obviously substantially prunes the search space of all possible variations on the circuit D. The filter provided by steps 2-6 then zeros in on the best description(s) of the faulty circuit.

9.3. Implementation in metaProlog

The full three-argument form of the demo predicate provides us with the facilities to accomplish this task. While the call (*) above will fail, the call

demo(D & laws, predict(If,Of), branch(B)) (9.7)

will succeed, binding B to an unsuccessful branch of the search tree. Thus the call

streamOf(B, demo(D&laws, predict(If,Of), branch(B)), Branches) (9.8)

will cause Branches to be bound to the list of all (failed) branches of the search tree. Having obtained this list, we then must sift through it to extract those branches which can be converted to successful proofs by changing the behavior of one pin on one gate.

To this end, it is would be convenient if all of the branches were organized so that all attempts to access facts in the gate portion (tt) of the circuit description D occurred last in the branch with no later processing of predict calls or topology calls. For then, the last (failed) goal on the branch will be a collection of gate-database (tt) calls, at least one of which fails. Filtering of the list Branches would then be easy, since we would only select those branches for which all but one of the gate-database (tt) calls in the final (failed) goal were in fact successful, and the modification to the gate database to make this branch successful is then obvious. That it is possible to so organize the generation of the search tree follows from one of the fundamental theorems at the foundation of logic programming, namely Hill's theorem to the effect that the existence of successful computations is independent of the rule for the choice of the next literal or call at each stage of exploration of the tree (cf. Lloyd[], p.). Thus we will utilize a computation rule which delays choosing "gate" calls (on tt) as long as possible: all calls on andGate and orGate (and other gates) will be pushed to the end of the branch. This control of the choice of the next literal at each stage of processing is achieved by use of

the "user_choice" control annotation in addition to the "branch(B)" annotation.

With these preliminaries, the top level of the diagnostic algorithm would now appear as:

```
all [Topo, Gates, I, O, Branches, HYPS, FAULTS] :
diagnose(c(Topo, Gates), p(I, O), FAULTS) <—
    streamOf(demo(Topo & Gates & laws,  predict(I, O),
                          branch(B)+user_choice), Branches) &
    make_hyps(Topo, Gates, I, O, Branches, HYPS) &
    test_and_elim(HYPS, FAULTS).
```

As indicated in Section 3, the "user_choice" control annotation causes the metaProlog interpreter, at each cycle of the basic deduction mechanism, to seek an assertion of the form

"user_choice(UC)"

in the theory under which it is carrying out the deduction. Recall that the theory tt above contains just such an assertion:

```
user_choice(delTab).
```

The interpreter expects delTab to define a predicate

choose(Goal, SubProblem, RemainingLiterals) .

The metaProlog interpreter tries to solve a call on this predicate relative to the theory delTab using the current main goal state in order to choose the next SubProblem of that main goal state for attempted resolution. Recalling the basic forms of goal statements from Section 4, we see that the following clauses will constitute an adequate definition of delTab:

```
all [B, Literal, Remainder] :
choose((true, B), Literal, Remainder) <—
    choose(B, Literal, Remainder).

all [A, B, Literal, RestB] :
choose((A, B), Literal, (RestB, A)) <—
    table(A) & choose(B, Literal, RestB).
```

```
all [A1, A2, B, Literal, RestA] :
choose(((A1, A2), B), Literal, (RestA, B)) <—
     choose((A1, A2), Literal, RestA).

all [A, B, Operator] :
choose((A, B), A, B) <—
     functor(A, Operator, _) &
     Operator \== ','.

all [A, B, Literal, Remaining, Operator] :
choose((A, B), Literal, Remaining) <—
     functor(A, Operator, _) &
     Operator \== ',' &
     choose(B, Literal, Remaining).

all [A, Operator] :
choose(A, A, true) <—
     functor(A, Operator, _) &
     Operator \== ','.

table(andTable(_, _, _, _)).
table(orTable(_, _, _, _)).
```

The predicate make_hyps simply recurses down the Branches list attempting to generate a candidate theory from the branch:

```
make_hyps(_, _, _, _, [], []).

all [Topo, Gates, I, O, Branch, Branches, Hyp, Hyps] :
make_hyps(Topo, Gates, I, O,  [Branch | Branches], [Hyp | Hyps] ) <—
     gen(Topo, Gates, I, O, Branch, Hyp) &
     make_hyps(Topo, Gates, I,O, Branches, Hyps).

all [Topo, Gates, I, O, Branches, HYPS] :
make_hyps(Topo, Gates, I, O, [_ | Branches], HYPS) <—
     make_hyps(Topo, Gates, I, O, Branches, HYPS).
```

The work of attempting to generate a candidate circuit description is carried out by the predicate "gen". For our purposes here, we will assume that the branch() control annotation has been implemented by what is in fact the convenient method of representing the branch as a

list in reverse order of generation, so that the last (failing) goal state is the head of the branch. Thus all gen needs to do is to pluck the head of the branch off, and determine whether or not it consists solely of "table" calls, all but one of which are in fact solvable relative to the theory Gates. If this is indeed the case, it will note the offending table call, say for gate G, and then generate the required theory from Gates by adding the assertion

exceptional(G).

to Gates, together with an explicit truth table for G (in the form of a collection of ___Table(G, ...) assertions) which is as close as possible to the standard table for gates of the type of G, but which allows the offending call to be solved. Since the gate G is now defined as being exceptional in Gates, the default standard rule for gates of the type of G will not succeed, but the explicit (non-standard) truth table for G will be used.

```
all [Topo, Gates, Goal, Hyp_Gates, Offendor] :
gen(Topo, Gates, [Goal | _], Topo & Hyp_Gates) <—
     single_f(Gates, Goal, Offendor) &
     mk_candidate(Gates, Offendor, Hyp_Gates).

all [Gates, TableCall, Rest] :
single_f(Gates, [TableCall | Rest], TableCall) <—
     not(demo(Gates, TableCall)) & all_work(Gates, Rest).

all [Gates, TableCall, Rest, Offendor] :
single_f(Gates, [TableCall | Rest], Offendor) <—
     demo(Gates, TableCall) & single_f(Gates, Rest, Offendor).

all_work(_, []).

all [Gates, TableCall, Rest] : all_work(Gates, [TableCall | Rest]) <—
     demo(Gates, TableCall) & all_work(Gates, Rest).

all [Gates, G, In1, In2, O, Hyp, Table] :
 mk_candidate(Gates, andTable(G, In1, In2, O), Hyp) <—
     mk_and_cand(Gates, G, In1, In2, O, Table) &
     add_all(Gates, [exceptional(G) | Table], Hyp).
```

```
all [Gates, G, In1, In2, O, Hyp, Table] :
mk_candidate(Gates, orTable(G, In1, In2, O), Hyp)  <—
     mk_or_cand(Gates, G, In1, In2, O, Table) &
     add_all(Gates, [exceptional(G) | Table], Hyp).

all [Gates, G, In1, In2, O, Rules] :
mk_and_cand(Gates, G, In1, In2, O, Rules)  <—
     Rules = [andTable(G, In1, In2, O),
               'all [I1, I2, OO]  :
                andTable(G, I1, I2, OO) <—
                    (I1 \== In1 ; I2 \== In2)   &
                      standardAnd(I1, I2, OO).' ]).

all [Gates, G, In1, In2, O, Rules] :
mk_or_cand(Gates, G, In1, In2, O, Rules)  <—
     Rules = [orTable(G, In1, In2, O),
               'all [I1, I2, OO]  :
                orTable(G, I1, I2, OO) <—
                   (I1 \== In1 ; I2 \== In2)   &
                    standardOr(I1, I2, OO).' ]).
```

In the last two rules, the quoted expression appearing as second element of the list in the body is just a shorthand for the present purposes. This shorthand indicates the result of building a term representing a rule using the appropriate naming operators. The last rule, for example, would more likely appear:

```
all [Gates, G, In1, In2, O, Rules] :
mk_or_cand(Gates, G, In1, In2, O, Rules)  <—
     Rules = [orTable(G, In1, In2, O), R_Others]   &
     mk_or_rule(G, In1, In2, O, R_Others).
```

The predicate mk_or_rule would use the naming operators to construct the indicated short-hand rule out of G, In1, In2, and O. [The obvious utility of the short-hand notation suggests a further extension of metaProlog allowing such constructs. Care in constructing such an extension must be exercised however, since a solution must be provided for the problem of quantifying into quotational contexts and all the referential opacity that would result.] Finally, add_all is defined as follows:

```
all T1 :
add_all(T1, [], T1).

all [T1, A, As, Result, T2] :
add_all(T1, [A | As], Result) <—
    addTo(T1, A, T2) &
    add_all(T2, As, Result).
```

9.4. Coroutining

The code for test_and_elim, while somewhat complex, is relatively straight-forward, and so will be omitted here. In this formulation of the diagnostic algorithm, the first

```
streamOf(demo(...
```

call, when run in a purely sequential version of metaProlog, produces a completed list of all branches of the search tree. This complete list is recursively processed by make_hyps, producing a completed list HYPS of candidates, and this is then recursively processed by test_and_elim. For real-world circuits, these lists might be unthinkably large. Instead of processing completed lists, it would be preferable to generate them as coroutined streams in a lazy manner, allowing consumption of branches by make_hyps as they are generated by the streamOf call, and allowing consumption of candidate theories by test_and_elim as they are generated by make_hyps.

The concurrency facilities of metaProlog will allow just such an approach. They in fact allow the organization of test_and_elim as a dynamically growing stream of filters, much as classical concurrent implementations of the seive of Eratosthenes. In the more general setting where we abandon the "single stuck-at" fault assumption, a given branch from the search tree may in fact produce more than one candidate theory. In this setting, we can use the concurrency facilities to organize make_hyp as a cascade of streams. This structure is indicated schematically in Figure B.9.2 below. In an implementation of metaProlog on a multiple-processor machine, the indicated processes could run concurrently on separate processors.

(fin,fout) - *faulty I/O pair*

demo(top&prop&fun, prp(fin,fout)) - *fails*

demo(top&prop&fun, prp(fin,fout), ??, Branches) - *succeeds*

failed branches of search space

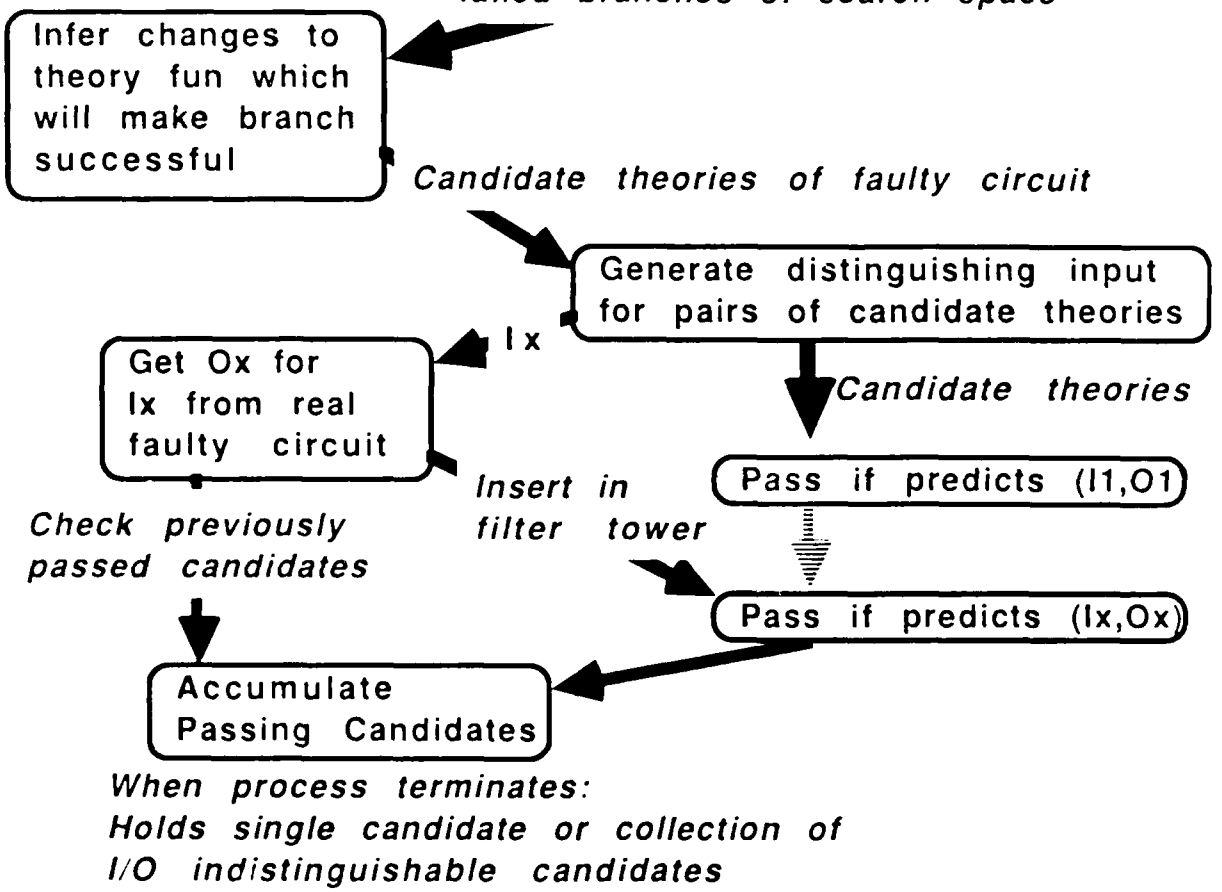


Figure B.9.2. Cascaded Generation & Testing of Candidate Theories.

9.5. Hierarchical Diagnosis

One classic cognitive technique for managing complexity is the imposition of hierarchical structure. Genesereth [] has observed that this approach can be of considerable use in the diagnosis of circuit faults. Here we will sketch the extension of the algorithm described above to a hierarchical setting.

Viewed hierarchically, complex devices can be seen as simple black boxes at one level, which, at the next lower level, decompose into collections of simpler devices. The connecting lines at the lower level may correspond directly to connecting lines at the upper level, or may themselves decompose into collections of simpler connecting lines. In the following diagram, the lines at the upper level decompose into collections of lines at the lower level.

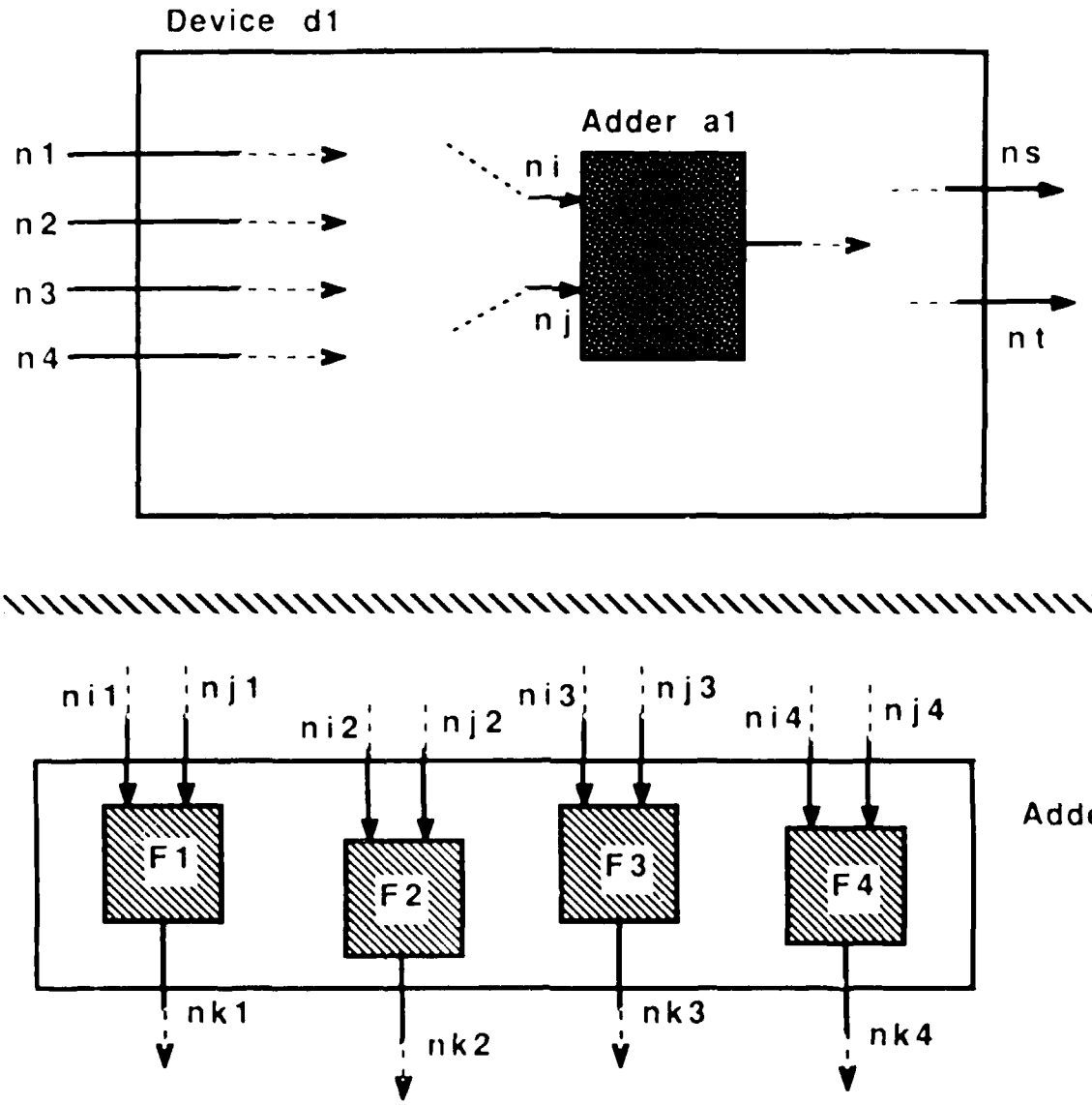


Figure B.9.3. Hierarchical Views of a Device.

While the diagram is simplified, at the upper level, the input lines might carry integers in the range 0,...256. These might correspond to collections of 8 lines at the lower level, each carrying a single bit. At each level of abstraction, we must have available theories which describe the topology of that level, the behaviors of individual components at that level, and the laws of propagation at that level (though these latter may not vary substantially from level to level). We will assume that devices are represented by a compound term of the form

d(Type, Topology, Behaviors, Laws, Corresp)

where Type is an atom indicating the kind of device, and Corresp is a collection of rules indicating the mapping from the input and output lines of the given device -- viewed as a black box at the upper level of abstraction -- to the internal lines when the device is "opened up" and viewed at the next lower level of abstraction. The collection of all such device descriptions is assumed to be stored in a named theory maintained at the top level of metaProlog, named "devs". [The system provides a "built in" predicate catalogue(Name,Theory) which is user-extensible.] Here is a sketch the revised hierarchical diagnostician:

```
all [Device,I,O,Topo,Behavs,Laws,Corresp,Branches,HYPS,Dev_FLTS]:
diagnose(Device, p(I,O)) <--
     catalogue(dev, d(Device, Topo, Behavs, Laws, Corresp)) &
     streamOf(B, demo(Topo&Behavs&Laws, predict(I,O),
               branch(B)+user_choice), Branches) &
     make_hyps(Topo, Behavs, I,O, Branches, HYPS) &
     test_and_elim(Device, HYPS, Dev_FLTS) &
     report_on(Device, Dev_FLTS) &
     decomp(Device, Dev_FLTS, p(I,O)).

all [Device, Hyp_List, IO_pair] :
decomp(Device, Hyp_List, IO_pair) <--
     primitive(Device) &
     report_on(Device, Hyp_List).

all [Device, IO_pair] :
decomp(Device, [], IO_pair) <--
     not(primitive(Device)).

all [Device, Hyp, Rest_Hyps, IO_pair, Device_Info,SubDevice,
          Sub_F_In, Sub_F_Out] :
decomp(Device, [Hyp | Rest_Hyps], IO_pair) <--
     not(primitive(Device)) &
     catalogue(dev, Device_Info) &
     ident_fault_comp(Device, Device_Info, IO_pair, SubDevice) &
     map_inputs(Device,Device_Info,SubDevice,IO_pair,Sub_F_In) &
     map_outs(Device, Device_Info,SubDevice,IO_pair,Sub_F_Out) &
     diagnose(SubDevice, p(Sub_F_In, Sub_F_Out)) &
     decomp(Device, Rest_Hyps, IO_pair).
```

10. Frames and Arrays

Frames (Minsky, FKL) are among the most widely used data structuring techniques in artificial intelligence programming. Abstractly, frames are just collections of slots with labels. Though they have many implementations, there appear to be two crucial properties common to most implementations:

i) The collection of slots making up an individual frame are physically grouped together in storage, guaranteeing that they can be accessed as a group. If the frame is a first class object, they can move about together.

ii) Besides being fillable with rather ordinary entities (atoms, numbers, compound expressions), certain slots can be filled with references to other frames. These references can be used to organize collections of frames in various kinds of hierarchies which can be exploited by systems which are utilizing the collection.

Typically, a frame carries an identifier which specifies it uniquely in the collection. Thus, a frame labelled "elephant" might contain the following slots among others:

a_kind_of : mammal
color : grey
number_of_toes : 4

Another frame, labelled "clyde" and intended to represent a particular elephant, might contain the following slots among others:

a_kind_of : elephant
home : london_zoo

The entry "elephant" in the frame for clyde might indeed be the identifier "elephant", or might be an internal direct reference to the frame representing generic elephants. In the former case, a frame processing system which was looking in clyde's frame and needed to obtain some information from the elephant frame (which is generic information generally true of all elephants) would first need to look up the identifier "elephant" in some internal table giving it the location of the generic elephant frame. Also, generic information is in such settings usually used

is a default, and can be over-ridden by information in the specific frame. Thus if clyde were an albino, the frame for clyde could contain the slot

color white

which would be accessed when clyde's color was needed. When attempting to obtain information from a particular slot in a particular frame, processing systems typically default to higher frames in the a kind of hierarchy only when the given slot is not present in the particular frame.

From a logical point of view (cf. [Hayes]), the slots of a frame and their contents can be viewed as assertions. Thus the frame for elephants could be taken to be equavalent to the collection

a_kind_of(elephant, mammal)
color(elephant, grey)
number_of_toes(elephant, 4)

while the frame for (the albino) clyde could be taken as being equivalent to the collection

a_kind_of(clyde, elephant)
home(clyde, london_zoo)
color(clyde, white).

But in metaProlog, the theory construct is designed specifically for the representation and manipulation of collections of assertions. Moreover, the collection of assertions making up a theory can indeed be physically grouped together, or else the actual arrangement is such that the access effects are very much as though the assertions were grouped together. Thus it is obviously natural in metaProlog to represent frames as (possibly small or large) theories containing assertions which correspond to the slots and their contents.

Given this point of view, it remains to be seen how we may organize the manipulation of these theories representing frames so as to achieve the effects produced by typical frame manipulation systems. (We will do this first from a straight-forward naive point of view; later we will refine the approach to achieve more compact storage utilization and the effects of direct embedded pointers from 'a kind_of' slots to other frames.)

New slot entries (i.e., new assertions) can be entered in the individual collection by use of the addTo built-in predicate. The question arises as to what transpires when one wishes to modify (update) an already existing slot value (i.e., in this setting remove an existing assertion and replace it by a new one). As will be discussed in Section 14, the implementation of theories is as a kind of "mutable array" which supports backtracking, yet provides all the speed of normal array access. In short, in a call

addTo(T1, A, T2),

the mutable array representing the theory to which the variable T1 is bound is actually updated by the insertion of A, this updated array is bound to T2, a descriptor referencing this updated array and A is bound to T1, and everything appropriate is trailed. The descriptor to which T1 is bound describes the original value of T1 in terms of A and the updated value currently bound to T2. See Section 14 for a more detailed discussion.

For our purposes here, it suffices to say that the theory representing the frame can be updated in such a way as to provide the same fast access as the original frame and yet still preserve the logical characater of the

...tation.

It remains to be seen just how the remaining actions of frame processors are effected in this context. This is most easily seen by considering the clauses which might be added to an ordinary Prolog definition of the metaProlog interpreter (i.e., the predicate demo). Briefly, one needs to add something like the following clauses to the set of clauses defining demo:

```
demo(Theory, Goal) :-
    Goal =.. [Predicate, Arg1 | RestArgs],
    demo(Theory, is_frame(Arg1, Frame_Theory)),
    demo(Frame_Theory, Goal).

demo(Theory, Goal) :-
    Goal =.. [Predicate, Arg1 | RestArgs],
    demo(Theory, is_frame(Arg1, Frame_Theory)),
    demo(Frame_Theory, a_kind_of(Arg1, B)),
    H =.. [Predicate, B | RestArgs],
    demo(Theory, is_frame(B, B_Frame_Theory)),
    demo(B_Frame_Theory, H).
```

As indicated above, we will later revised this to ... storage and increase efficiency. But for now, let us see ... work. Given the call

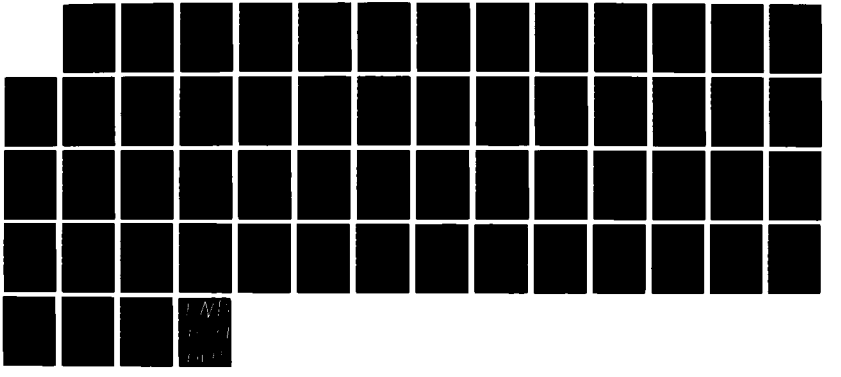
```
demo(animals, color(clyde, X))
```

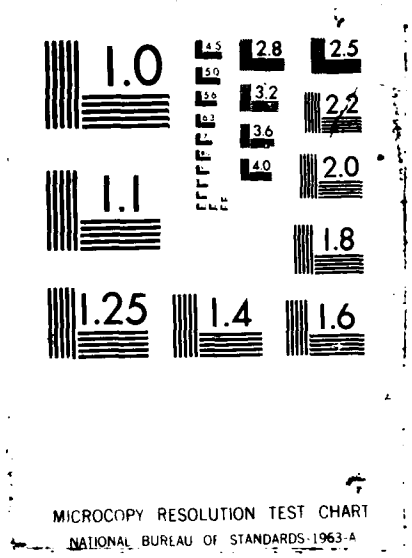
the first of the two clauses above will ... retrieved by the second call in ... effect of binding X to white ...

demo(anim...

the first ...

AD-A185 571 LOGIC PROGRAMMING AND KNOWLEDGE MAINTENANCE(U) SYRACUSE 2/2
UNIV NY SCHOOL OF COMPUTER AND INFORMATION SCIENCE
K A BOWEN 13 AUG 87 AFOSR-TR-87-1304 $AFOSR-82-0292
UNCLASSIFIED F/G 12/5 NL





will be run, finally binding N to 4, in the accepted manner of inheritance.

As naively described above, the organization of collections of frames is based on "master hash tables" containing pointers to the various frames. Given a modern workstation processor of sufficient power, this might not be a bad approach. However, the elements of the metaProlog approach provide a more sophisticated organization. The cost of the naive approach is encountered when following inheritance pointers such as "is_a_kind_of". The tracing of these pointers would involve multiple probes in the master hash table. However, these can be replaced by the tracing of pointer chains as follows.

The notion of a metaProlog name is similar to the natural language notion of name. It is a syntactic item which somehow directly refers to the object it names. The details of an implementation method are irrelevant so long as the name relation posseses the required abstract properties. Consequently, we are free to implement the name relation and names any way which provides the essential "referring" property of names. Since all the items referred to by metaProlog names are themselves syntactic items, the things named are ultimately just computer data structures which must reside at locations in memory. Consequently, a prime candidate for the implementation of names is the use of internal memory pointers referring to the locations of the data structures. Most likely these references will be more complex than raw pointers — for example, they might be tagged pointers. But metaProlog names are just metaProlog entities, no different in general character from other metaProlog entities, and consequently, names can participate in assertions just like all other entities. Hence, the inheritance assertions contained in a frame can refer to the super-ordinate frame using such a name of the super-ordinate frame. But then, following inheritance frames simply involves extracting the memory address from the name, and following the resulting pointer, etc.

11. Truth Maintenance & Concurrency

11.1. Related metaProlog Facilities.

Part of the original motivation for the original design of metaProlog was the desire to provide a logically sounder basis for the use of logic databases. Specifically, many Prolog programs utilize the program database itself to represent knowledge being manipulated by the program. This involves on-the-fly modification of the program database by the built-in predicates assert and retract. This has at least three drawbacks:

1. It is logically unsound. There is no known logical basis for performing deductions from a set of axioms which vary as the proof is being constructed: classical logical is a-temporal.

2. There exists the possiblity of confusion between the program itself and the knowledge base it manipulates since they occupy the same name space.

3. There is no possibility of dealing directly with distinct alternative knowledge bases, since everything must be recorded in the single program database.

Sequential metaProlog solves these problems. While sequential metaProlog is logically sound, some difficulties remain in the run-time interpretation of some constructs. To deal with these difficulties, we explored borrowing constructs from Concurrent Prolog. Besides solving the problems, the concurrency constructs permit a useful programming style for reasoning systems.

Ordinary Prolog systems provide for the representation of just one logical theory: the program in its entirety is identified with that theory. Yet there are many circumstances in which one would find it extremely useful to be able to represent different logical theories within the same program. This facility could be used, for example, in a medical diagnosis and therapy program not only to increase modularity and efficiency by segregating information about different classes of diseases and drugs into different theories, but also to represent alternative diagnostic and therapeutic approaches and regiemes.

metaProlog can be thought of as being obtained by starting with an

ordinary Prolog system and extending it so that the following criteria are met:

1) For every term and formula E of the system, there is a term of the system, say call it

n(E),

which serves as a name of E. The connection between the two is provided by a primitive predicate

name(E,n(E)).

2) For every finite set S of formulas of the system, there is a term t(S) which is thought of as the name of the theory whose axioms are the members of T. The connection between the members of S and t(S) is provided by the primitive predicate

axiom_of(X,t(S))

which holds between X and t(S) iff X is the name of an element of S.

3) There is a primitive predicate

demo(T, G, C)

which holds iff T is the name of a theory, G is the name of a goal, and C describes generalized control information to be obeyed in searching for a proof of G from T.

4) There exist primitive predicates

addTo(T, F, U) and dropFrom(T, F, U)

such that if T and U are names of theories and F is the name of a formula, then:

addTo(T, F, U) holds iff U names the theory obtained from the theory named by T by means of adding the formula named by F as an axiom,

dropFrom(T, F, U) holds iff U names the theory obtained from the theory named by T by means of removing the formula named by F from the axiom set of T (and doing nothing if it does not occur there.)

Besides use of the predicate addTo described in 4), unions of two theories such as T and S can be implicitly referenced via calls such as

demo(T & S, G, C, P).

11.2. Simple Reason Maintenance

Consider a simple reason maintenance system designed to record the answers of suspects being questioned by our detective Poirot as described in Section 5. Assume that reasons are maintained in a theory called TM, and that evidence for an assertion is maintained by the predicate

evidence_for(Assertion, Reasons)

where Reasons is simply a list of the supporting evidence for Assertion. The assertions themselves are recorded in a theory called KB. Suspects, being nothing more than the sum total of their beliefs, are represented by the theory consisting of their beliefs. For example, john (in a recasting of the original Rosie version) consists of:

```
needs(john,money).
married_to(john,mary).
loves(john,mary).
```

(mary is the dead victim in this thriller.) The victim's sister sara consists of:

```
sister(sara,mary).
loves(sara,john).
false(loves(john,mary)).
loves(john,sara).
```

(Note that sara believes that john loves her, while john's feelings differ somewhat.) Questions to suspects are generated by the user. Given a question Q, the system poses the question to a suspect, say john, by running

the goal

?-demo(john, Q).

Depending on the success or failure of this goal, the system passes either q_a(john,yes) or q_a(john,no) together with Q to the reason maintenance predicate rm, which can be defined by the clauses:

```
all [KB,TM,Q,Who,KB,NewTM] :
rm(KB, TM, Q, q_a(Who,yes),KB, NewTM) <—
  demo(KB, Q) &
  addEvidence(TM, Q, q_a(Who,yes),NewTM).

all [KB,TM,Q,Who,KB,NewTM] :
rm(KB, TM, Q, q_a(Who, yes),NewKB, NewTM) <—
  demo(KB, false(Q)) &
  revise(KB, TM, false(Q) &
  q_a(Who, yes) &
  NewKB, NewTM).

all [KB,TM,Q,Who,KB,NewTM] :
rm(KB, TM, Q, q_a(Who, yes),NewKB, NewTM) <—
  addTo(KB, Q, NewKB) &
  addEvidence(TM, Q, q_a(Who, yes) &
  NewTM).
```

Similar clauses must be added for combinations such as false(Q) with q_a(Who, yes), Q with q_a(Who, no), etc. The predicate addEvidence is defined by:

```
all [TM,Q,Reason,NewTM,New_evidence,Intermed_TM] :
addEvidence(TM, Q, Reason, NewTM) <—
  demo(TM, evidence_for(Q, Evidence)) &
  dropFrom(TM, evidence_for(Q,_)), Intermed_TM) &
  insert(Reason, Evidence, New_Evidence) &
  addTo(Intermed_TM, evidence_for(Q, New_Evidence),NewTM).
```

The relevant clause for revise is:

```
all [TM,Q,Reason,NewTM,NewKB,Pos_Ev,Neg_Ev,Concl] :
revise(KB, TM, false(Q),Reason, NewKB, NewTM) <—
  demo(TM, evidence_for(Q, Pos_Ev)) &
  demo(TM, evidence_for(false(Q),Neg_Ev)) &
  insert(Q, Neg_Ev, Neg_Ev) &
  demo(resolve, adjudicate(KB, false(Q),Pos_Ev, Neg_Ev, Concl)) &
  finish_rev(KB, TM, false(Q), Concl, Pos_Ev, Neg_Ev, NewKB,
NewTM).
```

The predicate finish_rev is similar in spirit to addEvidence. More interesting is the theory resolve which encodes rules for resolving contradictions according to the evidence for an assertion and its negation. Packaging this as a separate theory allows such a system to be easily adjusted for varying applications. It even allows differing theories of conflict resolution to be selected dynamically by the system according to criteria depending on the structure of Q, or on criteria to be found in KB or TM (and thus possibly varying in time).

In this case, resolve is simple:

```
all [KB,TM,What,Ev_For,Ev_Against] :
adjudicate(KB,TM,What,Ev_For,Ev_Against,preserve) <—
  member(q_a(Who,yes), Ev_For) &
  demo(KB, reliable(Who)).

all [KB,TM,What,Ev_For,Ev_Against,SomeOne] :
adjudicate(KB,TM,What,Ev_For,Ev_Against,reverse) <—
  member(q_a(Who,yes),Ev_Against) &
  demo(KB, reliable(Who)) &
  not((member(q_a(SomeOne, yes),Ev_For) &
  demo(KB, reliable(SomeOne))).

all [KB,TM,What,Ev_For,Ev_Against,LFor,LAg] :
adjudicate(KB,TM,What,Ev_For,Ev_Against,preserve) <—
  length(Ev_For, LFor) &
  length(Ev_Against, LAg) &
  LAg =< LFor.
```

```
all [KB,TM,What,Ev_For,Ev_Against,LFor,LAg] :
adjudicate(KB,TM,What,Ev_For,Ev_Against,reverse) <—
  length(Ev_For, LFor) &
  length(Ev_Against,LAg) &
  LAg > LFor.
```

Here member is ordinary list membership, while length is ordinary list length, and not is the common 'negation by failure' of logic programming.

11.3. Adding Concurrency to metaProlog

One can conceive of starting with either metaProlog and adding concurrency features, or conversely, beginning with a system such as Concurrent Prolog, and adding metaProlog features as system built-ins. We adopt the latter approach here. Thus, theories are regarded as first-class objects, as earlier, and demo is treated as a built-in from the point of view of Concurrent Prolog. However, unlike the other built-ins, it is backtrackable, so that among other things, it provides an interface from the Concurrent Prolog interpreter to a sequential interpreter. If a theory argument is added to the Concurrent Prolog interpreter "solve" of Shapiro [ ], then the interface back from the sequential world to the concurrent world is provided by allowing calls of the form

```
demo(Theory, Goal, concurrent).
```

This causes Goal to be solved by a Concurrent Prolog interpreter which carries Theory as its additional argument. In our current experimental system, the user surface level is regarded as Concurrent Prolog running against a user-supplied theory as its extra argument. The upper-level of the detective program considered earlier now appears as follows:

```
all [TerminalInput,Poirot_KBM,Poirot_User,
            KBM_User,TerminalOutput] :
detective <—
    instream(TerminalInput) &
    poirot(TerminalInput?, Poirot_KBM, Poirot_User, KBM_User?) &
    kbm(TerminalInput?, Poirot_KBM?, KBM_User) &
    merge(Poirot_User?, KBM_User?, TerminalOutput) &
    outstream(TerminalOutput?).
```

Beyond the user, the major components of this program are the knowledge base manager, kbm, which records the results of the questioning together with maintenance of the reasons, and a detective, poirot, who listens to the questioning and attempts to make deductions regarding the suspects. The kbm is defined by:

```
all [TermIn,Poirot_KBM,KBM_User] :
    kbm(TermIn, Poirot_KBM, KBM_User )<—
        kbm(TermIn, Poirot_KBM, KBM_User,common &
        integ(integ) & tm(tm)).

kbm([], _, [], _).


all [Query,TermIns,Poirot_KBMs,KBM_Users,KB] :
    kbm([q(Query) | TermIns], Poirot_KBMs,
                    [true(Query) | KBM_Users], KB) <—
        demo(KB, Query) |
        kbm(TermIns?, Poirot_KBMs, KBM_Users, KB).

all [Query,TermIns,Poirot_KBMs,KBM_Users,KB] :
    kbm([q(Query) | TermIns], Poirot_KBM,
                    [false(Query) | KBM_Users], KB) <—
        demo(KB, false(Query)) |
        kbm(TermIns?, Poirot_KBMs, KBM_Users, KB).

all [Query,TermIns,Poirot_KBMs,KBM_Users,KB] :
    kbm([q(Query) | TermIns], Poirot_KBM,
                    [unknown(Query) | KBM_Users], KB) <—
        otherwise |
        kbm(TermIns?, Poirot_KBMs, KBM_Users, KB).

all [Query,TermIns,Poirot_KBMs,KBM_Users,KB,Who,
                            What,NewKB,Maint_Resp] :
    kbm([ask(Who,What) | TermIns],Poirot_KBMs,
                    [answer(Who,What,Response) | KB_Users], KB) <—
        question(Who, What, Response) &
        reason_maint(KB, What,
                    q_a(Who,Response?),NewKB, Maint_Resp) |
        kbm(TermIns?, Poirot_KBMs, KB_Users, NewKB).
```

```
all [Evid,TermIns,Poirot_KBMs,KBM_Users,KB,TM,What] :
     kbm([evidence(What) | TermIns], Poirot_KBMs,
               [evidence(What, Evid) | KB_Users], KB) <—
          demo(KB,tm(TM)) &
          demo(TM?,evidence_for(What, Evid))
          |  kbm(TermIns?, Poirot_KBMs, KB_Users, KB).

all [Command,TermIns,Poirot_KBMs,KBM_Users,KB] :
     kbm([Command | TermIns],Poirot_KBMs,
               [unknown_cmd(Command) | KB_Users], KB) <—
          otherwise |
          kbm(TermIns?, Poirot_KBMs, KB_Users, KB).


all [Who, What] :
     question(Who, What, yes) <—
          demo(Who & common, What)
          | true.

all [Who, What] :
     question(Who, What, no) <—
          otherwise
          | true.

all [KB,What,Reason,NewKB,TM,InterKB,NewTM,MidKB] :
     reason_maint(KB, What, Reason, NewKB) <—
          writeInl(['Init Reason Maint: ', What, ' - ',Reason])
           |  demo(KB, tm(TM)) &
          rm(KB, TM?, What, Reason, InterKB, NewTM) &
          dropFrom(InterKB?, tm(TM),MidKB) &
          addTo(MidKB?, tm(NewTM),NewKB).
```

It remains to sketch the definition of the detective poirot who listens to the questions asked (by having access to the streams TerminalInput and KBM_User) and who attempts to make deductions based on the evidence. Since the kbm is intended to implement the corporate detective memory, the simplest version of Poirot provides him with no local memory (i.e., private theory) of his own, but forces him to rely on the kbm with which he interacts through the stream Poirot_KBM. (Additional clauses must be added to the definition of kbm to reflect the interaction; we will indicate

some of these as we proceed.)

```
poirot([], [], [], _).

all [Who,What,RestQ,P_KBM,P_User,KBM_User,Q] :
     poirot([ask(Who, What) | RestQ], P_KBM, P_User, KBM_User) <—
          perk_up(Who, What, P_KBM, P_User, KBM_User?) &
          poirot(RestQ, P_KBM, P_User, KBM_User).

all [Who,What,,P_KBM,P_User,KBM_User,
                         Message,Rest_KBM_User,Response] :
     perk_up(Who, What, P_KBM, P_User, [Message | Rest_KBM_User])
      <—
          Message = answer(Who, What, Response) |
          try_deduction(Who, What, Response, P_KBM, P_User).


all [Who,What,,P_KBM,P_User,KBM_User,
                         Message,Rest_KBM_User,Response] :
     perk_up(Who, What, P_KBM, P_User, [Message | Rest_KBM_User])
      <—
     otherwise |
     perk_up(Who, What, P_KBM, P_User, Rest_KBM_User?).

all [Who,What,Response,KB,P_KBMs,PUser,Whom] :
     try_deduction(Who, What, Response, [cur(KB) | P_KBMs], P_User)
     <—
          demo(relevance, concerns(What, Who, Response, Whom)) |
          try_suspect(KB, Whom, P_KBMs, P_User).

 try_deduction(_, _, _, _, _) <—
     otherwise | true.
```

```
all [KB,Who,Motive,Proof,P_KBMS,P_Users] :
    try_suspect(KB, Who,
                [record(suspect(Who, Motive),Proof) | P_KBMs],
                [poirot(suspect(Who, Motive)) | P_Users])
    <—
    demo(suspect & KB, suspect(Who, Motive),prolog, Proof) |   true.

try_suspect(_, _, _, _) <—
    otherwise | true.
```

The additional necessary clauses for kbm are:

```
all [TermIn,KB,Poirot_KBMs,KBM_User,KB] :
    kbm(TermIn, [cur(KB) | Poirot_KBMs], KBM_User, KB) <—
        kbm(TermIn, Poirot_KBMs?, KBM_User, KB).

all [TermIn,KB,Poirot_KBMs,KBM_User,KB,
                        Assertion,Reason,Asserions,Reasons] :
    kbm(TermIn,
            [record(Assertion,Reason) | Poirot_KBMS], KBM_User, KB)
    <—
        reason_maint(KB, Assertion, Reasons, NewKB) &
        kbm(TermIn, Poirot_KBMS?, KBM_User, NewKB?).
```

Note that since theories are first-class objects, poirot uses the paritally instantitated message cur(X) on the stream Poirot_KBM to request the entire current state of the knowledge base from the kbm, and use it in his deductions.

The theory relevance contains rules for concluding when a given question and response leads to a concern regarding a possible suspect (Whom), while suspect is Poirot's theory of what makes a person a suspect with what motive. It appears as follows:

```
    all [Person,OtherPerson,Victim] :
        suspect(Person, jealousy) <—
            loves(Person, OtherPerson) &
            married(OtherPerson, Victim) &
            found_dead(Victim).
```

```
all [Person,Victim] :
     suspect(Person, greed) <—
          need(Person, money) &
          found_dead(Victim) &
          rich(Victim) &
          related(Person, Victim).

all [Person,OtherPerson,Victim] :
     suspect(Person, revenge) <—
          loved(OtherPerson, Victim) &
          not(OtherPerson = Person) &
          found_dead(Victim) &
          rejected_by(Person, OtherPerson).

all [Person,OtherPerson] :
     rejected_by(Person, OtherPerson) <—
          loves(Person, OtherPerson) &
          not(Person = OtherPerson) &
          not(loves(OtherPerson, Person)).
```

Appendix to Section 11:
The Combined metaProlog/Concurrent Simulator.

```
:-op(1199,xfy,'|').
:-op(450, xf, '?').
:-op(950, xfy, '&').

c :-  op(1199,xfy,'|'),
      op(450,xf,'?').

watch :-
      retract(value(trace,off)).

nowatch :-
      assert(value(trace,off)).

setup :-
      set(smode,depth_first),
      set(traceset,[calldemo(_),sucdemo(_),
            demodemo(_),reduction(_),suspension(_)]),
      set(countingset,[]),
      set(smode(read(_)),breadth_first),
      set(initialized,true).

trc(Form) :-
       clause(value(traceset,Current),true,PTR),
      erase(PTR),
      assert(value(traceset,[Form | Current])).
      tr(Pred/Arity) :-

source_clause(Pred,Arity,_),
      functor(Form,Pred,Arity),
      trc(call(Form)),
      trc(reduction(Form)).

solve(Goal) :-
      clear_counters,
      solve(Goal, 0),
      display_counters.

solve(true, _) :-
!.
solve(otherwise,_) :-!.

solve([otherwise | Rest], Depth) :-
      solve(Rest, Depth).

solve(Goal, Depth) :-
      con_system(Goal),!,
      trace(system(Depth), Goal), Goal;
        trace(solve(Depth),Goal),
```

```
        schedule(Goal, X,X, Head, [cycle(1) | Tail]),
        solve(Head, Tail, nodeadlock, Depth),
       trace(solved(Depth),
       Goal).

solve([otherwise | Head], Tail, DL, D) :-!,
      solve(Head, Tail, DL, D).

solve([cycle(N)], _, _, D) :-!,
      (D=0, write(['***cycles: ',N]), nl; true).

solve([cycle(N) | Head], [], deadlock, D) :-!,
      D=0, write(['***cycles: ',N]),nl,
      writelnl(['***Deadlock detected. Locked processes:' | Head]);   fail.

solve([cycle(N) | Head], [cycle(N1) | Tail], nodeadlock, D) :-
      N1 is N + 1,
      solve(Head, Tail, deadlock, D).

solve([read(X) | Head], Tail, DL, D) :-
      solve_wait_writes(Head, Tail, DL, D,
                              NewHead, NewTail, NewDL, NewD),
      read(X),
      solve(NewHead, NewTail, nodeadlock, NewD).

solve([(A & B) | Head], Tail, DL, D) :-!,
      D1 is D+1,
       solve(A, D1),
      solve(B, D1), !,
      solve(Head, Tail, DL, D).

solve([A | Head], Tail, DL, D) :-
      con_system(A),!,
      trace(system(D),A),
       A,
       solve(Head, Tail, nodeadlock, D).

solve([demo(T,A) | Head], Tail, DL, D) :-!,
      solve([demo(T,A,prolog,[],_) | Head], Tail, DL, D),!.

solve([demo(T,(A,B),prolog,InPrf,OutPrf) | Head], Tail, DL, D) :-
      D1 is D+1,
      trace(calldemo(D1),T/A),
        (built_in_meta(A), A, !,Reas=bi(A) ;
          retrieve(T,A),
          Reas=s(A,fact)),
      trace(sucdemo(D1), T/(A/true)),
      schedule(demo(T, B, prolog, [Reas | InPrf], OutPrf),
                        Head, Tail, NewHead, NewTail),
      solve(NewHead, NewTail, nodeadlock,D1).

solve([demo(T, (A,B),prolog, InPrf, OutPrf) | Head], Tail, DL, D) :-
      D1 is D+1,
      trace(calldemo(D1), T/A),
      retrieve(T, (A <— Body)),
```

```
        trace(sucdemo(D1), T/(A <—Body)),
        schedule(demo(T, (Body, B),prolog, [s(A,(A<—Body))| InPrf], OutPrf),
                        Head, Tail, NewHead, NewTail),
        solve(NewHead, NewTail, nodeadlock,D1).

solve([demo(T1, demo(T2, A), C, InPrf, OutPrf) | Head], Tail, DL, D) :-
        D1 is D+1,
        trace(demodemo(D1),T1/(T2/A)),
        schedule(demo(T2, A, C, [], SubPrf),
                        Head, Tail, NewHead, NewTail),
        solve(NewHead, NewTail, nodeadlock, D1).

solve([demo(T, A, C, InPrf, OutPrf) | Head], Tail, DL, D) :-
        D1 is D+1,
        trace(calldemo(D1),T/A),
        (built_in_meta(A),!,A,Reas=bi(A),Flag=ok ;
          con_retrieve(T, A, Flag),
          Reas = s(A,fact)),
        (Flag=ok,!,trace(sucdemo(D1),T/A),
             solve(Head, Tail, nodeadlock, D1),
            OutPrf=[Reas | InPrf];
         Flag=susp,
              schedule(suspended(demo(T,A,C,InPrf,OutPrf)),Head,Tail,NH,NT),
            solve(NH, NT, DL, D1)).

solve([demo(T, A, C, InPrf, OutPrf) | Head], Tail, DL, D) :-
        D1 is D+1,
        trace(calldemo(D1),A),
        con_retrieve(T, (A <— B),Flag),
        (Flag=ok,!,
          trace(sucdemo(D1),T/(A <—B)),
          schedule(demo(T, B, C, [s(A, (A <—B)) | InPrf], OutPrf),
                        Head, Tail, NewHead, NewTail);
         Flag=susp,
          schedule(suspended(demo(T,A,C,InPrf,Outprf)),
                                    Head,Tail,NewHead, NewTail)),
        solve(NewHead, NewTail, nodeadlock, D1).

solve([demo(_,_,_,_,_)|Head],Tail,Dl,D) :-!,fail.

solve([A | Head], Tail, DL, D) :-
        D1 is D+1,
        trace(call(D1),A),
        reduce(A, B, DL, DL1, D1),
        trace(reduction(D1),(A<—B)),
        schedule(B, Head, Tail, NewHead, NewTail),!,
        solve(NewHead, NewTail, DL1, D).

solve_wait_writes(Head, Tail, DL, D, Head, Tail, DL, D) :-
        Head == Tail.
```

```
solve_wait_writes([wait_write(X,Y) | Head], Tail, DL, D,
                        NewHead, NewTail, NewDL, NewD) :-
      not(var(X)),!,
      call(write(Y)),
      nl,
      solve_wait_writes(Head, Tail, DL, D,
                              NewHead, NewTail, NewDL, NewD).

solve_wait_writes([Process | Head], Tail, DL, D,
                        [Process | NewHead], NewTail, NewDL, NewD) :-
      solve_wait_writes(Head, Tail, DL, D,
                              NewHead, NewTail, NewDL, NewD).


reduce(demo(_,_),_,_,_,_) :-!,fail.
reduce(demo(_,_,_,_,_),_,_,_,_) :-!,fail.

reduce(A, B, _, nodeadlock, D) :-
      guarded_clause(A, G, B, D),
      trace(try_clause(D),(A<—(G | B))),
      solve(G, D),
      !.

reduce(A, suspended(A), DL, DL, D) :-
      trace(suspension(D),A).

reduce(demo(T,true,_,InPrf,InPrf),true,_,nodeadlock,D).

reduce(A, B) :-
      guarded_clause(A,G,B,1),
      solve(G,1).

reduce(A, suspended(A)) :-
      trace(suspension(A)).

schedule(true, Head, Tail, Head, Tail) :-!.
schedule(suspended(A),Head, [A | Tail], Head, Tail) :-!.

schedule((A,B),Head, Tail, NewHead, NewTail) :-
      value(smode, breadth_first),
      !,
      schedule(A, Head, Tail, Head1, Tail1),
        schedule(B, Head1, Tail1, NewHead, NewTail),
      !.

schedule((A,B),Head, Tail, NewHead, NewTail) :-
      value(smode, depth_first),
      !,
      schedule(B, Head, Tail, Head1, Tail1),
      schedule(A, Head1, Tail1, NewHead, NewTail),
      !.

schedule(A, Head, Tail, [A | Head], Tail) :-
      value(smode(A),
      depth_first),
```

```
      !.

schedule(A, Head, [A | Tail], Head, Tail) :-
      value(smode(A),
      breadth_first),
      !.

schedule(A, Head, Tail, [A | Head], Tail) :-
      value(smode, depth_first),
      !.

schedule(A, Head, [A | Tail], Head, Tail) :-
      value(smode, breadth_first),
      !.

guarded_clause(A, G, B, D) :-
      ready_clause(A, B1, D),
      find_guard(B1, G, B).

find_guard((A | B),A, B) :-!.
find_guard(A, true, A).


ready_clause(A, B, D) :-
      functor(A, F, N),
      functor(A1, F, N),
      clause(A1, B),
      race(concurrent_unify(D),(A, A1)),
       concurrent_unify(A, A1).

concurrent_unify(X,Y) :-
      (var(X) ; var(Y) ), !, X = Y.
concurrent_unify(X?, Y) :-!,
      nonvar(X),
      concurrent_unify(X, Y),
      !.
concurrent_unify(X, Y?) :-!,
      nonvar(Y),
      concurrent_unify(X, Y),
      !.
concurrent_unify([X | Xs], [Y | Ys]) :-!,
      concurrent_unify(X, Y),
       concurrent_unify(Xs, Ys),
      !.
concurrent_unify([], []) :-!.

concurrent_unify(X, Y) :-
      X =..[F | Xs],
      Y =..[F | Ys],
      concurrent_unify(Xs, Ys),
      !.
concurrent_unify(X,Y,ok) :-
      (var(X); var(Y)),
       !, X=Y.
```

```
concurrent_unify(X?, Y, susp) :-
      var(X),
       nonvar(Y),
      !.
concurrent_unify(X?, Y, Flag) :-!,
      nonvar(X),
      concurrent_unify(X, Y, Flag),
      !.
concurrent_unify(X, Y?, susp) :-
      nonvar(X),
      var(Y),
      !.
concurrent_unify(X, Y?, Flag) :-!,
      nonvar(Y),
      concurrent_unify(X, Y, Flag),
      !.
concurrent_unify([X | Xs], [Y | Ys], Flag) :-!,
      concurrent_unify(X, Y, F1),
      (F1=susp,!, Flag=susp;
        nonvar(F1),
        F1=ok,concurrent_unify(Xs, Ys, F2),
       (F2=susp,!, Flag=susp;
           nonvar(F2),
           F2=ok,!,Flag=ok)),
      !.

concurrent_unify([], [], ok) :-!.

concurrent_unify(X, Y, Flag) :-
      X =.. [F | Xs],
      Y =.. [F | Ys],
      concurrent_unify(Xs, Ys, Flag),
      !.

trace(_, _) :-
      value(trace, off),
       !.

trace(A, B) :-
      add_counter(A),
      % break(A, B),        % add a break package
      value(traceset, S),
      (member(A,S); S = all),
      writel([A, ':', B]),
       nl, !.
trace(_, _).


clear_counters :-
      value(counter(X),Y),
      Y > 0,
      set(counter(X),0), fail; true

add_counter(A) :-
      value(countingset, S),
```

```
        member(A, S),
        add1(counter(A), _); true.

display_counters :-
        value(countingset, S),
        member(X, S),
        value(counter(X),Y),
        Y > 0,
        writel(['# ',X,': ',Y]),
        nl,fail;
          sum_counters;
          true.

sum_counters :-
        value(countingset, S),
        setof(Y, X^(member(X, S),value(counter(X),Y)),1),
        sum(S1, 0, Total),
        writel(['Total: ', Total]), nl.

sum([],Temp,Temp).

sum([H | T],Temp,Total) :-
        NT is H+Temp,
        sum(T,NT,Total).

strip_qs(X,X) :-
        var(X),
        !.
strip_qs(X?,Y) :-
        !,   strip_qs(X,Y).

strip_qs([],[]) :-
        !.
strip_qs([H | T],[SH | ST]) :-
        !,
        strip_qs(H,SH),
        strip_qs(T,ST).

strip_qs(In,Out) :-
        In =..[F | Args],!,
         strip_qs(Args, Stripped_Args),
        Out =..[F |Stripped_Args].

strip_qs(X,X).

set(A,B) :-
        (clause(value(A,V),true,PTR),
           !,erase(PTR);
         true),
        assert(value(A,B)).

wait(X) :-
        wait(X, _).

wait(X, _) :-
        var(X),
```

```
      !,fail.

wait(X?, Y) :-
      !, wait(X, Y).

wait(X, X).



dif(X, Y) :-
      (var(X); var(Y)),
      !,fail.
dif(X?, Y) :-
      !,  dif(X, Y).

dif(X, Y?) :-
      !,  dif(X, Y).

dif([], []) :-
      !, fail.

dif([X | Xs], [Y | Ys]) :-
      !,
      dif(X, Y);
        dif(Xs, Ys).

dif(X, Y) :-
      X =..[Fx | Xs],
      Y =.. [Fy | Ys],
      (Fx \== Fy ;
        dif(Xs, Ys)).

dif(X, Y) :-
      (var(X); var(Y)),
       fail.

con_system(wait(_, _)).
con_system(wait(_)).
con_system(dif(_,_)).
con_system(fread(_)).
con_system(otherwise).
on_system(writel(_)).

%deal with the meta built-ins

con_system(addTo(_,_,_)).
con_system(dropFrom(_,_,_)).
con_system(consult(_,_)).
con_system(ask(_,_)).
con_system(Otherwise) :-
      system1(Otherwise).

all Xs :
instream(Xs) <—
      read(X) | instream(X, Xs).
```

```
instream(end_of_file, []).

instream(close_stream, []).

all [Xs,Ys,Y] :
instream([], Xs) <—
      instream(Y?, Xs),
      read(Y).

all [X,Xs,Ys] :
instream([X | Xs], [X | Ys]) <—
      instream(Xs, Ys).

all [X,Xs,Y] :
instream(X, [X | Xs]) <—
      wait(X) |
      instream(Y?, Xs),
      read(Y).

all [X,Xs] :
outstream([X | Xs]) <—        %Xs is the current output stream
      writel(['*** outstream: ', X]), nl |
      outstream(Xs?).
      outstream([]).

all [X,Y] :
wait_write(X,Y) <— % wait for X and output Y to current output stream
      wait(X) |    call((write(Y),nl)).

% wrap stream elements with an identifying tag

wrap([], _, []).
all [X,Xs,WrappedX,Ys,W] :
wrap([X | Xs], W, [WrappedX | Ys]) <—
      WrappedX =..[W, X] | wrap(Xs?, W, Ys).

all [X,X1,Y,Y1] :
lt(X,Y) <—
      wait(X, X1),
      wait(Y, Y1) | X1 < Y1.

all [X,X1,Y,Y1] :
le(X,Y) <—
      wait(X,X1),
      wait(Y, Y1) | X1 =< Y1.

% lazy evaluator or arithmetic expressions
eval(X,Y) :-
      wait(X,Y), integer(Y) | true.
eval(X+Y,Z) :-
      eval(X?, X1), eval(Y?,Y1), plus(X1, Y1, Z).
eval(X-Y,Z) :-
      eval(X?, X1), eval(Y?,Y1), plus(Z, Y1, X1).
eval(X*Y,Z) :-
```

```
      eval(X?, X1), eval(Y?,Y1), times(X1, Y1, Z).

plus(X,Y,Z) :-wait(X,X1), wait(Y,Y1) | Z is X1+Y1.
plus(X,Y,Z) :- wait(X,X1),wait(Z,Z1) | Y is Z1-X1.
plus(X,Y,Z) :- wait(Y,Y1),wait(Z,Z1) | X is Z1-Y1.

times(X,Y,Z) :- wait(X,X1),  wait(Y,Y1) | Z is X1*Y1.
times(X,Y,Z) :-  wait(X,X1), wait(Z,Z1) | Y is Z1/X1.
times(X,Y,Z) :- wait(Y,Y1), wait(Z,Z1) | X is Z1/Y1.

member(X,[X | _]).
member(X, [Y | T]) :- member(X, T).

writel(X) :- var(X),!.
writel([]) :-!.
writel([H | T]) :- write(H), !, writel(T).

writelnl([]) :- nl,!.
writelnl([H | T]) :- write(H), nl,!, writelnl(T).

fread(X) :- write('>>'), read(X).

system1((X is Y)).
system1(true).
system1(ancestors(Anc1)).
system1(call(X)).
      ...
      ...



con_retrieve(Theory, (Call <—Body),Flag) :-
      functor(Call, F, N),
      functor(Call1, F, N),
      retrieve(Theory, (Call1 <—Body)),
      trace(concurrent_unify(D),(Call, Call1)),
      concurrent_unify(Call, Call1, Flag).

con_retrieve(Theory, Call, Flag) :-
      functor(Call, F, N),
      functor(Call1, F, N),
      retrieve(Theory, Call1),
      trace(concurrent_unify(D),(Call, Call1)),
      concurrent_unify(Call, Call1, Flag).
```

12. A metaProlog Simulator.

```
/*-------------------------------------------------------------------------
    File: demo_react.pro
    Author: Kenneth A. Bowen
    Date:  24 July 1985

    Notes: Central interpreter for metaProlog
-------------------------------------------------------------------------*/

:-op(1000, xfy, '&').
:-op(1150, xfy, '&&').
:-op(1100, xfy, '<--').
:-op(1100, xfy, ':').
:-op(1101, fy, all).
:-op(990, fy, if).
:-op(985, xfy, then).
:-op(980, xfy, else).

demo(Theory, Goal)
    :—
     demo(Theory, Goal, Control).

/*
demo(Theory, Goal, [])
    :-
    empty(Goal).
    */

demo(Theory, true, [])
    :—!.

demo(Theory, Goal, [Reason | Rest_Proof])
    :-
    select(Goal, SubGoal, Rest_Goals, Theory),
    react(Theory, SubGoal, Reason, Continuation_Goals),
    merge(Continuation_Goals, Rest_Goals, New_Goal, Theory),
    demo(Theory, New_Goal, Rest_Proof).


react(Theory, demo(New_Theory, Subsid_Goal, Subsid_Proof),
                    sbs(Subsid_Proof), true)
    :- !,
    demo(New_Theory, Subsid_Goal, Subsid_Proof).

react(Theory, demo(New_Theory, Subsid_Goal), sbs(Subsid_Proof), true)
    :- !,
    demo(New_Theory, Subsid_Goal, Subsid_Proof).

react(Theory, current(Theory), [current], true):—!.

react(Theory, (Vars : Goal), strip_vars, Internal_Goal)
    :- !,
```

```
    make_internal((Vars : Goal), Internal_Goal, true).

react(Theory, not(Goal), neg(Goal), true)
    :-
    \+(demo(Theory, Goal, _)).

react(Theory, (if Condition then SuccessGoal else FailureGoal),
                    if_then_else(s(Condition, Cond_Proof)), SuccessGoal)
    :-
    demo(Theory, Condition, Cond_Proof), !.

react(Theory, (if Condition then SuccessGoal else FailureGoal),
                    if_then_else(f(Condition)), FailureGoal)
    :- !.

react(Theory, (if Condition then Goal), if_then(s(Condition, Cond_Proof)), Goal)
    :-
    demo(Theory, Condition, Cond_Proof), !.

react(Theory, Goal, built_in(Goal), true)
    :-
    built_in(Goal), !,
    Goal.

/*  --- Additions for frame  processing....

react(Theory, Goal, fr(Frame_Trace), true)
    :-
    Goal =.. [Pred, Arg1 | Rest_Args],
    is_theory(Arg1),
    Frame_Goal =.. [Pred | Rest_Args],
    demo(Arg1, Frame_Goal, Frame_Trace).

react(Theory, Goal, inh(Proof), true)
    :-
    demo(Theory, is_a(Super_Frame_Name), _),
    name_of(Super_Frame_Name, Super_Frame_Theory),
    demo(Super_Frame_Theory, Goal, Proof).

react(Theory, update(Frame, Slot, New_Value), upd(Frame, Slot, New_Value), true)
    :-
    Old_Assert =..[Slot, Old_Value],
    drop_from(Frame, Old_Assert, Intermed_Frame),
    New_Assert =.. [Slot, New_Value],
    add_to(Intermed_Frame, New_Assert, New_Frame).
*/

/* Modified form to use to allow demon processing on update; similar
     modification should be made to other frame axioms if demon processing
     is desired there; e.g., on access, or on inheritance, etc.....

react(Theory, update(Frame, Slot, New_Value), upd(Frame, Slot, New_Value), true)
    :-
    Old_Assert =..[Slot, Old_Value],
    drop_from(Frame, Old_Assert, Intermed_Frame_0),
```

```
    New_Assert =.. [Slot, New_Value],
    add_to(Intermed_Frame, New_Assert, Intermed_Frame_1),
    demo(Intermed_Frame_1,
        demon(Slot, Old_Value, New_Value, Intermed_Frame_1), _).
*/

react(Theory, send(Destination_Theory, Message, Response),
            send(Destination_Theory, Message, Response), true)
    :-
    demo(Destination_Theory, receive(Message, Response), _).

react(Theory, assert(Database_Theory, Assertion, Response),
        assert(Database_Theory, Assertion, Response), true)
    :-
    demo(Database_Theory,
        process(add(Assertion, Response),
                Database_Theory, New_Database_Theory), _).

react(Theory, SubGoal, s(SubGoal, Rule), Rule_Body)
    :-
    find(SubGoal, Theory, Rule),
    parts(Rule, Rule_Head, Rule_Body),
    match(SubGoal, Rule_Head).



find(Goal, Theory_U/Theory_V, Clause)
    :-
    retrieve(Theory_U, subtheory(Theory_V), true, _),
    find(Goal, Theory_V, Clause).

find(Goal, Theory, (Goal:— Body))
    :-
    retrieve(Theory, Goal, Body, _).

parts((Head:— Body), Head, Body).

match(Item, Item).

select( ((SubSubGoal & SubSubGoals) & SubGoals),
            SubSubGoal, (SubSubGoals & SubGoals), _):—!.

select((SubGoal & SubGoals), SubGoal, SubGoals, _):— !.

select((SubGoal && SubGoals), SubGoal, bf(SubGoals), _):— !.

select(Goal, Goal, true, _).

merge(New_SubGoals, true, New_SubGoals, _):—!.

merge(true, Continuation, Continuation, _):— !.

merge(New_SubGoals, Continuation, (New_SubGoals & Continuation), _).

check_demo_spying_enter(Goal) :—
```

```
    spy_or_trace(Goal),
    write('demo:enter: '),write(Goal),nl.

check_demo_spying_enter(Goal):—no_spy_or_trace(Goal).

spy_or_trace(Goal):—demo_spying(Goal).

spy_or_trace(Goal):—demo_tracing.

no_spy_or_trace(Goal):—
    \+(demo_spying(Goal)), \+(demo_tracing).

check_demo_spying_exit(Goal):—
    spy_or_trace(Goal),
    write('demo:exit: '),write(Goal),nl.

check_demo_spying_exit(Goal):—
    spy_or_trace(Goal),
    write('demo:retry: '), write(Goal),nl, !, fail.

check_demo_spying_exit(Goal):—no_spy_or_trace(Goal).

demo_trace:— assert(demo_tracing).
demo_notrace:— retract(demo_tracing).
demo_notrace.


/*-------------------------------------------------------------------------
    File: meta_top_level.pro
    Author: Kenneth A. Bowen
    Date: 20 May 1985
-------------------------------------------------------------------------*/
m:—
    start_up_meta.

start_up_meta
    :-
    $prompt(2, '', '|:'),
      write('-----------------------------------'),nl,
    write('metaProlog 0.5'),nl,
    write('(c) 1985 Kenneth A. Bowen'),nl,
    write('All rights reserved.'),nl,nl,
    write('system file? '),
     read(System_File),
     consult_and_go(System_File),
     write('after consult_and_go succeed:..aborting to Prolog'),nl,
     abort.

start_up_meta
    :—
     abort.

consult_and_go(System_File)
    :—
     metaProlog_consult(System_Theory, System_Id, System_File),
     write('File '),write(System_File),write(' consulted to theory '),
```

```
    write(System_Theory), nl,
    demo(System_Theory, system_startup,                                    Prf).


metaProlog_consult(System_Id, System_Id, System_File)
    :—
     write('metaConsult:file='),write(System_File),nl,
    (var(System_Id), !, theory_gensym(System_Id); true),
     see(System_File),
     read_and_record(System_Id),
     close(System_File),
     asserta(parent_theory(System_Id, empty_theory, consult)).

read_and_record(System_Id)
    :—
     read(Item), !,
     dispatch_read_and_record(Item, System_Id).

dispatch_read_and_record(end_of_file, _):— !.

dispatch_read_and_record(Item, System_Id)
    :—
     make_internal(Item, Head, Body),
     assertz((Head:— Body), Ref),
     assertz(belongs_to(Ref, System_Id)),
     !,
     read_and_record(System_Id).


/*-------------------------------------------------------------------------
    File: module_db_mgr.pro
    Author: Kenneth A. Bowen
    Date 19 May 1985
        Module-based revisions begun 27 Aug 85
-------------------------------------------------------------------------*/

is_theory('$th.%'(_, _, _)).

retrieve(empty_theory, Goal, Body, Control):— fail.

retrieve((T1 & T2), Goal, Body, Control)
    :—
       retrieve(T1, Goal, Body, Control) ; retrieve(T2, Goal, Body, Control).

retrieve(T, Goal, Body, Control)
    :—
    clause(Goal, Body, Reference),
    belongs_to(Reference, T).

belongs_to(Reference, T)
    :—
    parent_of(T, T1, _),
    belongs_to(Reference,T1).

/* Ground "belongs_to" assertions are created by add_to with asserta */
```

```
add_To(Theory, Assertion, New_Theory_Id)
    :— !,
     make_internal(Assertion, Head, Body),
     theory_gensym(New_Theory_Id),
     assertz((Head:— Body), Ref),
     asserta(belongs_to(Ref, New_Theory_Id)),
     asserta(parent_theory(New_Theory_Id, empty_theory, add(Assertion))).

instance((Vars : Goal), Internal_Goal, Internal_Variables)
    :— !,
     create_variables(Vars, Internal_Variables),
     replace(Goal, Vars, Internal_Variables, Internal_Goal).

instance(( Goal/Vars ), Internal_Goal, Internal_Variables)
    :— !,
     instance((Vars : Goal), Internal_Goal, Internal_Variables).

instance(Goal, Goal, []).

make_internal(all(Item), New_Head, New_Body)
    :- !,
    make_internal(Item, New_Head, New_Body).

make_internal((Vars : (Head <-- Body)), New_Head, New_Body)
    :— !,
     create_variables(Vars, Internal_Vars),
     replace(Head, Vars, Internal_Vars, New_Head),
     replace(Body, Vars, Internal_Vars, New_Body).

make_internal((Vars : Fact), New_Fact, true)
    :— !,
     create_variables(Vars, Internal_Vars),
     replace(Fact, Vars, Internal_Vars, New_Fact).

make_internal((Head <-- Body), Head, Body):— !.

make_internal(Fact, Fact, true):— !.

create_variables([], []).

create_variables([Identifier | Rest_Identifiers], [Var | Rest_Vars])
    :— !,
     create_variables(Rest_Identifiers, Rest_Vars).


replace((A & B), Vars, Internal_Vars, (New_A & New_B))
    :— !,
     replace(A, Vars, Internal_Vars, New_A),
     replace(B, Vars, Internal_Vars, New_B).

replace([], Vars, Internal_Vars, []).

replace([First | Rest], Vars, Internal_Vars, [New_First | New_Rest])
```

```
    :— !,
     replace(First, Vars, Internal_Vars, New_First),
     replace(Rest, Vars, Internal_Vars, New_Rest).

replace(A, _, _, A)
    :—
     integer(A), !.

replace(A, Vars, Internal_Vars, New_A)
    :—
     atom(A), !,
     look_up(Vars, A, Internal_Vars, New_A).

replace(A, Vars, Internal_Vars, New_A)
    :—
     A =..[Operator | Args], !,
     replace(Args, Vars, Internal_Vars, New_Args),
     New_A =..[Operator | New_Args].



look_up([], A, _, A):— !.

look_up([A | _], A, [New_A | _], New_A):— !.

look_up([_ | Rest_Vars], A, [_ | Rest_Internal_Vars], New_A)
    :— !,
     look_up(Rest_Vars, A, Rest_Internal_Vars, New_A).



'@# % % '(0).

theory_gensym(M)
    :—
     retract('@# % % '(N)),
     M is N+1,
     assert('@# % % '(M)).

append([], X, X).
append([H | T], Y, [H | Z])
    :-
        append(T, Y, Z).

bagOf(Template, Goal, Output)
    :—
     gensym(Tag),
     assert('%bag% Of% store % '(Tag, [])),!,
     '%bag O%f '(Tag, Template, Goal, Output).

'%bag O%f '(Tag, Template, Goal, Output)
    :—
     Goal, add_element_bagof(Tag, Template), fail.

'%bag O%f '(Tag, Template, Goal, Output)
```

```
    :—
     '%bag% Of% store % '(Tag, Temp),
     retract('%bag% Of% store % '(Tag, _)),
     reverse(Temp, Output).

add_element_bagof(Tag, Template)
    :—
     '%bag% Of% store % '(Tag, Current),
     retract('%bag% Of% store % '(Tag, Current)),
     assert('%bag% Of% store % '(Tag, [Template | Current])), !.

reverse(X, X).

gensym(Tag)
    :—
    retract('@g#e$n%s^y&m*'(gensym(N))),  !,
     M is N+1,
     Tag = gensym(M),
    assert('@g#e$n%s^y&m*'(gensym(N))).

gensym(gensym(0))
    :—
    assert('@g#e$n%s^y&m*'(gensym(0))).

/*-------------------------------------------------------------------------
    File: system_predicates.pro
    Author: Kenneth A. Bowen
    Date : 19 May 1985
-------------------------------------------------------------------------*/

built_in(true).
built_in(fail).
built_in(update(_,_,_)).
built_in(add_To(_, _, _)).
built_in(drop_From(_, _, _)).
built_in(metaProlog_consult(_,_,_)).
built_in(read(_)).
built_in(write(_)).
built_in(nl).
built_in(atom(_)).
built_in(instance(_, _, _)).
built_in(halt).
built_in(abort).
built_in(save(_)).
built_in((spy _)).
built_in((nospy _)).
built_in(trace).
built_in(notrace).
built_in(demo_trace).
built_in(demo_notrace).
built_in(demo_spy(_)).
built_in(deep_spy(_)).
built_in(no_demo_spy(_)).
built_in(no_deep_spy(_)).
built_in(bagOf(_, _, _)).
built_in(set_slot(_, _, _)).
```

```
built_in(make_frame(_)).
built_in((_ < _)).
built_in((_ =< _)).
built_in((_ > _)).
built_in((_ >= _)).
built_in(( _ is _)).
built_in((_ = _)).
built_in(built_in(_)).
```

12.2. Towards a More Serious Simulator (by Keith Hughes).

The following is a description of the current metaProlog simulator. The idea is different from the previous metaProlog simulator in that clauses are now fully compiled, rather than interpreted (at least as far as I understand the previous implementation). This compilation is achieved through the guard clause, an idea proposed by Andy Turk for compiling meta at the machine level.

The basic idea is that theories are denoted by lists. When an addTo or dropFrom is done, a unique theory ID is generated. The path to this new theory is then the new theory ID appended to the end of the old theory descriptor. In the case of an addTo, a new clause is asserted, while in the case of a dropFrom, the old clause in the database is modified.

For example, if the fact p is added to the empty theory ([]), a new theory descriptor of [0] would be created (addTo([],p,[0])). A rewritten form of p is then asserted into the database. This new form has an extra argument added to the beginning of the head and to each subgoal in the body of a clause. For instance, in the case of p above, the new clause asserted would be

```
p(TheoryIn) :— guard(TheoryIn,[0|_],[]).
```

guard does the real work in meta. It makes sure that the clause can be used in TheoryIn, the theory descriptor handed to the call of p by demo. The second argument of guard describes the theories in which p is known by giving the initial sequence of all theories that can access p. For each theory that cannot access p, the third argument of guard contains the initial sequence of all theories where the clause is no longer valid. This third argument will be a list of lists. A clause with aguments and subgoals, such as

```
p(A) :— g(A,B), r(B)
```

would be asserted into theory [0,1] as

```
p(TheoryIn,A) :—
     guard(TheoryIn,[0,1|_],[]),
    g(TheoryIn,A,B),r(TheoryIn,B).
```

The theory behind guard is simple. If a sequence of addTo's is done, the theory descriptor coming out of the last addTo is a list of that theory. For example,

:— addTo([],p,T1),addTo(T1,q,T2),addTo(T2,r,T3)

would instantiate T1 to [0], T2 to [0,1], and T3 to [0,1,2]. p is known in all of these theories, and the second argument of it's guard clause is [0|_], which would unify with any of the above theory descriptors (T1,T2,T3). [0|_] is said to be an initial sequence of these theory descriptors. So, any theory descriptor starting with a 0 knows about p. q's initial sequence is [0,1|_]. This is not an initial sequence of [0], so cannot be found in the first theory, but will be found in theories [0,1] and [0,1,2].

dropFrom works by adding to the third argument of a clause. To continue the above example, if a dropFrom([0],p,T4) is executed, T4 will be instantiated to [0,3], and the clause in the database will be changed to

p(TheoryIn) :— guard(TheoryIn,[0|_],[[0,1,2|_]]).

So, this third argument contains a list of initial sequences of theory descriptors with roots where p was defined (theory [0] in this case), where p no longer exists. If p was put into meta from another addTo, this addTo would create a different clause for p of the same form as above. The guard clause, if the first two arguments unify, will check TheoryIn against the theory descriptors in the list in the third argument position, and fail guard if any are an initial sequence of TheoryIn.

guard also handles the union of theories. If the user does a demo(T1+T2+...+Tn,Goals), the entire group of T1+T2+....+Tn is passed through to the Goals in the first argument. If guard notices a union, it will try and find the clause in the first theory in the list. If it isn't there, the next theory will be tried. This checking is put into guard to avoid excess work. If a late subgoal in the list of Goals fails to be found in T1, the interpretor must not fail all the previous subgoals just because this subgoal can't find anything in T1. So, this subgoal will look in T2 next.

demo(TheoryID,Goals) will call Goals with a particular TheoryID. The list of goals is rewritten to pass in TheoryID as the first argument. If TheoryID is a variable, demo will unify TheoryID to a theory descriptor

where the goal is true. addTo and dropFrom are implemented so that they can backtrack. If a cut is executed, when backtracking is done, the clauses will be left in the database. This is not a problem, however, since the guard for these clauses will never fire again.

There are two extra builtins added for fun. One is called baseTheory(FileName), which is like consult. The file contains lines like

theory(TheoryName).
theory for TheoryName..... .....
endTheory.
theory(NewTheoryName).
etc.

Once a theory is described, clauses can also be added by

TheoryName :: Clause.

To use clauses in a theory entered using baseTheory, the predicate theory(TheoryName,TheoryID) is used in conjunction with demo. For instance,

....., theory(phideaux,T1), demo(T1,Goal), ...

would pick up the theory ID bound to the name phideaux and use it in demo. These theory names cannot themselves be used by the core meta predicates.

An example of a file for baseTheory follows.

```
----------
theory(a).
a(a). a(b). a(c). a(d).
endTheory.
theory(b).
b(a). b(b). b(c). b(d).
endTheory.
b :: b(e). a :: a(e) :— a(b).
----------
```

```
/*-----------------------------------------------------------------------------
    demo(TheoryID,Goals) will call Goals with a particular TheoryID. If TheoryID
    is a variable, demo will unify TheoryID to a theory descriptor where the goal is
    true.

    addTo and dropFrom are as they should be; they even backtrack. There are two
    extra builtins I added for stuff I'm doing. One is called

        baseTheory(FileName),

    which is like consult. the file contains lines like

    theory(TheoryName).
    theory for TheoryName..... .....
    endTheory.
    theory(NewTheoryName).
    etc.

    Once a theory is described, clauses can also be added by

    TheoryName :: Clause.

    For an example, see the file ~hughes/research/meta/test.  To use clauses in a
    theory entered  using baseTheory, the predicate theory(TheoryName,TheoryID)
    is used in conjunction with demo. for instance,

    ....., theory(relativity,T1), demo(T1,Goal), .....

    would pick up the theory  ID bound to the name relativity and use it in demo.
    These theory names cannot themselves be used by the core  meta predicates.

    The reader doesn't understand variable quantification yet; standard prolog
    rules are used.

    The dropFrom bug is fixed. Clauses really disappear when they are supposed to. It
    makes the code run a little slower when there are a lot of dropFroms, but that's
    life.
    -----------------------------------------------------------------------------*/


% written by keith hughes  4/10/86
%
% modified by ken bowen 4/28/86 --

:—consult('../system_predicates.pro').

:— op(1200,xfy,::).
:—op(1000, xfy, '&').
:—op(1150, xfy, '&&').
:—op(1100, xfy, '<-').
:—op(1100, xfy, ':').
:—op(1101, fy, all).
:—op(990, fy, if).
:—op(985, xfy, then).
:—op(980, xfy, else).
```

```
%
% guard(CurTheory,BottomNode,WhereIAm,ExclusionList):
% CurTheory is where we currently are in the execution state at this time.
% BottomNode is the last element in the list CurTheory
% WhereIAm is the theory in which the clause was compiled into
% ExclusionList is a list of the bottom elements of a theory list where
%   the clause no longer exists


guard(Theory,CurTheory,ExclusionList) :-
   nonvar(Theory),
   Theory =FirstTheory+RestTheories,
    guard(FirstTheory,CurTheory,ExclusionList).
guard(Theory,CurTheory,ExclusionList) :-
   nonvar(Theory),
   Theory =FirstTheory+RestTheories,
    guard(RestTheories,CurTheory,ExclusionList).
guard(CurTheory,CurTheory,ExclusionList) :-
   notDropped(ExclusionList,CurTheory).

notDropped(Element,_) :-
   var(Element),!.
notDropped([],_) :- !.
notDropped([E1 | _],E2) :-
  initial(E1,E2),
  !, fail.
notDropped([_ | Rest],E) :-
  notDropped(Rest,E).

initial(E1,E2) :-
   var(E2),!,fail.
initial(E1,E2) :-
  var(E1),!.
initial([X | Rest1],[X | Rest2]) :-
   initial(Rest1,Rest2).

%
% addTo(OldTheory,Clause,NewTheory): add Clause to OldTheory, creating
%   NewTheory
%

addTo(CurTheory,Clause,NewTheory) :-
    newTheoryDesc(CurTheory,NewTheory,End),
    adjustClause(Clause,NewTheory,NewClause),
    asserta(clauses_of(NewTheory,NewClause)),
   xassert(NewClause,DBRef),
  End = [],   % yuch gasp argh
    backtrackAddTo(DBRef).

exhibit(Theory) :-
   clauses_of(Theory,Clause),
   show_clause(Clause),
  fail.
exhibit(Theory).
```

```
show_clause(C) :— write(C),nl.
%
% this backtracks addTo
%

backtrackAddTo(DBRef).
backtrackAddTo(DBRef) :—
  $dbref_erase(DBRef),
  fail.


%
% just append, sort of
%

newTheoryDesc([],[NewID | End],End) :— !,
  newTheoryID(NewID).
newTheoryDesc([X | L1],[X | L2],End) :—
  newTheoryDesc(L1,L2,End).

currentTheoryID(0).

newTheoryID(X) :—
  currentTheoryID(X),
  retract(currentTheoryID(X)),
  NX is X + 1,
  assert(currentTheoryID(NX)).

adjustClause((Head :— Tail),TheoryID,(NewHead :— Guard,NewBody)) :— !,
  adjustHead(Head,NewHead,TheoryIn),
  adjustBody(Tail,NewBody,TheoryIn),
  makeGuard(TheoryID,TheoryIn,Guard).
adjustClause(Head,TheoryID,(NewHead :— Guard)) :—
  adjustHead(Head,NewHead,TheoryIn),
  makeGuard(TheoryID,TheoryIn,Guard).

makeGuard(TheoryID,TheoryIn,guard(TheoryIn,TheoryID,[])).

adjustHead(Head,NewHead,TheoryIn) :—
  Head =.. [Functor | Args],
  NewHead =.. [Functor,TheoryIn | Args].

adjustBody((addTo(T1,Goal,T2),Rest),(addTo(T1,Goal,T2),NewRest),TheoryID) :— !,
  adjustBody(Rest,NewRest,TheoryID).
adjustBody((demo(T,Goal),Rest),(demo(T,Goal),NewRest),TheoryID) :— !,
  adjustBody(Rest,NewRest,TheoryID).
adjustBody((dropFrom(T1,Goal,T2),Rest),
      (dropFrom(T1,Goal,T2),NewRest),TheoryID) :— !,
  adjustBody(Rest,NewRest,TheoryID).
adjustBody((First,Rest),(First,NewRest),TheoryID) :—
  built_in(First), !,
  adjustBody(Rest,NewRest,TheoryID).
adjustBody((First,Rest),(NewFirst,NewRest),TheoryID) :— !,
  First =.. [Functor | Args],
```

```
   NewFirst =.. [Functor,TheoryID | Args],
   adjustBody(Rest,NewRest,TheoryID).
adjustBody(addTo(T1,Goal,T2),addTo(T1,Goal,T2),_) :— !.
adjustBody(demo(T,Goal),demo(T,Goal),_) :— !.
adjustBody(dropFrom(T1,Goal,T2),dropFrom(T1,Goal,T2),_) :— !.
adjustBody(Only,Only,TheoryID) :—
   built_in(Only),!.
adjustBody(Only,NewOnly,TheoryID) :—
  Only =.. [Functor | Args],
  NewOnly =.. [Functor,TheoryID | Args].

%
% dropFrom(OldTheory,Clause,NewTheory)
%

dropFrom(TheoryID,Clause,NewTheoryID) :—
   newTheoryDesc(TheoryID,NewTheoryID,NewEnd),
   fixClause(Clause,TheoryID,End,NewTheoryID),
  NewEnd = [].    % yech arg garg

fixClause((Head :— Body),TheoryID,End,NewID) :— !,
   adjustHead(Head,NewHead,TheoryIn),
   adjustBody(Body,NewBody,TheoryIn),
    clause(Head,(guard(TheoryIn,WhereIAm,Exclude),NewBody),ADBRef),
   not(not(guard(TheoryID,WhereIAm,Exclude))), % horrider and horrider
   $dbref_erase(ADBRef),
  xassert((NewHead :—
guard(TheoryIn,WhereIAm,[NewID | Exclude]),NewBody),DBRef),
   backtrackDropFrom(DBRef,
              (NewHead :— guard(TheoryIn,WhereIAm,Exclude),NewBody)).
fixClause(Head,TheoryID,End,NewID) :—
   adjustHead(Head,NewHead,TheoryIn),
    clause(NewHead,guard(TheoryIn,WhereIAm,Exclude),ADBRef),
   not(not(guard(TheoryID,WhereIAm,Exclude))), % horrider and horrider
   $dbref_erase(ADBRef),
    xassert((NewHead :— guard(TheoryIn,WhereIAm,[NewID | Exclude])),DBRef),
    backtrackDropFrom(DBRef,((NewHead :— guard(TheoryIn,WhereIAm,Exclude)))).

%
% this is so dropFrom can backtrack
%

backtrackDropFrom(DBRef,Clause).
backtrackDropFrom(DBRef,Clause) :—
  $dbref_erase(DBRef),
  assert(Clause),
  fail.

%
% demo(Theories,Goals)
%

demo(Theories,Goals) :—
   var(Theories),!,
   rewriteGoals(Goals,Theories,NewGoals),
   call(NewGoals),
```

```
   fixTheory(Theories).
demo(Theories,Goals) :—
   rewriteTheory(Theories,NewTheories),
   rewriteGoals(Goals,NewTheories,NewGoals),
   call(NewGoals).

% needed to fill in last element part of theory description

fixTheory([LastElement]) :— !.
fixTheory([_|RestElements]) :—
   fixTheory(RestElements).

%
% add in theory ID carriers
%

rewriteGoals((addTo(T1,Goal,T2),RestGoals),Theories,
        (addTo(T1,Goal,T2),NewRestGoals)) :— !,
   rewriteGoals(RestGoals,Theories,NewRestGoals).
rewriteGoals((demo(T,Goal),RestGoals),Theories,
        (demo(T,Goal),NewRestGoals)) :— !,
   rewriteGoals(RestGoals,Theories,NewRestGoals).
rewriteGoals((dropFrom(T1,Goal,T2),RestGoals),Theories,
        (dropFrom(T1,Goal,T2),NewRestGoals)) :— !,
   rewriteGoals(RestGoals,Theories,NewRestGoals).
rewriteGoals((setOf(T,Gs,L),RestGoals),Theories,
        (setOf(T,NGs,L),NRestGoals)) :—!,
   rewriteGoals(Gs,Theories,NGs),
   rewriteGoals(RestGoals,Theories,NRestGoals).
rewriteGoals((FirstGoal,RestGoals),Theories,(FirstGoal,NewRestGoals)) :—
   built_in(FirstGoal),!,
   rewriteGoals(RestGoals,Theories,NewRestGoals).
rewriteGoals((FirstGoal,RestGoals),Theories,(NewFirstGoal,NewRestGoals)) :— !,
  FirstGoal =.. [Functor|Args],
   NewFirstGoal =.. [Functor,Theories|Args],
   rewriteGoals(RestGoals,Theories,NewRestGoals).
rewriteGoals(demo(T,Goal),Theories,demo(T,Goal)) :— !.
rewriteGoals(addTo(T1,Goal,T2),Theories,addTo(T1,Goal,T2)) :— !.
rewriteGoals(dropFrom(T1,Goal,T2),Theories,dropFrom(T1,Goal,T2)) :— !.
rewriteGoals(setOf(T,Gs,L),Theories, setOf(T,NGs,L)) :—!,
   rewriteGoals(Gs,Theories,NGs).
rewriteGoals(OnlyGoal,Theories,OnlyGoal) :—
   built_in(OnlyGoal),!.
rewriteGoals(OnlyGoal,Theories,NewOnlyGoal) :—
  OnlyGoal =.. [Functor|Args],
   NewOnlyGoal =.. [Functor,Theories|Args].

%
% baseTheory(File): read in a base theory
%

baseTheory(File) :—
  see(File),
  procBaseTheory,
  seen.
```

```
procBaseTheory :—
  read(Clause),
   procBaseTheory(Clause).

procBaseTheory(end_of_file) :— !.
procBaseTheory(theory(TheoryName)) :— !,
   newTheoryID(NewID),
   assert(theory(TheoryName,[NewID])),
   write(TheoryName),write(' = '), write([NewID]),nl,
   readTheoryClauses([NewID|_]),
  procBaseTheory.
procBaseTheory((TheoryName :: Clause)) :—
   theory(TheoryName,TheoryID),
   fixID(TheoryID,NTheoryID),
   addTo(NTheoryID,Clause),
  procBaseTheory.

readTheoryClauses(TheoryID) :—
  read(Clause),
   procTheoryClause(TheoryID,Clause).

procTheoryClause(_,end_of_file) :— !.
procTheoryClause(_,endTheory) :— !.
procTheoryClause(_,stable(Fact)) :—!,
  assert(stable(Fact)),
   readTheoryClauses(TheoryID).
procTheoryClause(_,Clause) :—
  stable(Clause),!,
   make_internal(Clause, Head, Body),
  enter(Head,Body),
   readTheoryClauses(TheoryID).
procTheoryClause(TheoryID,Clause) :—
   make_internal(Clause, Head, Body),
   addTo(TheoryID,(Head :— Body)),
   readTheoryClauses(TheoryID).

stable(stable(_)).
stable((Head <- Body)) :— !, stable(Head).
stable((all Vars : Clause)) :— !, stable(Clause).
enter(Head, true) :— !,assertz(Head).
enter(Head, Body) :— assertz((Head :— Body)).
fixID([ID],[ID|_]) :— !.
fixID([ID|Rest],[ID|ERest]) :—
  fixID(Rest,ERest).

%
% addTo(TheoryID,Clause): add Clause to TheoryID with no new ID
%

addTo(TheoryID,Clause) :—
   adjustClause(Clause,TheoryID,NewClause),
  assert(NewClause).

%
```

```
% rewriteTheory(OldTheory,NewTheory): rewrite virtual theory descriptor
% for demo
%

rewriteTheory(OldTheory,OldTheory) :—
   var(OldTheory),!.
rewriteTheory(OldTheory-Clause,NewTheory) :— !,
   subtractClause(OldTheory,Clause,NewTheory).
rewriteTheory(OldTheory+OldTheory2,NewTheory+NewTheory2) :— !,
   rewriteTheory(OldTheory,NewTheory),
   rewriteTheory(OldTheory2,NewTheory2).
rewriteTheory(OldTheory,NewTheory) :—                    %take care of named theories
   theory(OldTheory,NewTheory),!.
rewriteTheory(OldTheory,OldTheory).

subtractClause(FrontTheory+RestTheories,Clause,
         NewFrontTheory+NewRestTheories) :— !,
   subtractClause(FrontTheory,Clause,NewFrontTheory),
   subtractClause(NewRestTheories,Clause,NewRestTheories).
subtractClause(Theory,Clause,NewTheory) :—
   rewriteTheory(Theory,NTheory),
   dropFrom(NTheory,Clause,NewTheory).

instance((Vars : Goal), Internal_Goal, Internal_Variables)
  :— !,
    create_variables(Vars, Internal_Variables),
    replace(Goal, Vars, Internal_Variables, Internal_Goal).

instance(( Goal/Vars ), Internal_Goal, Internal_Variables)
  :— !,
   instance((Vars : Goal), Internal_Goal, Internal_Variables).

instance(Goal, Goal, []).

make_internal(all(Item), New_Head, New_Body)
    :— !,
    make_internal(Item, New_Head, New_Body).

make_internal((Vars : (Head <- Body)), New_Head, New_Body)
  :— !,
   make_list(Vars, LVars),
   create_variables(LVars, Internal_Vars),
   replace(Head, LVars, Internal_Vars, New_Head),
   replace(Body, LVars, Internal_Vars, New_Body).

make_internal((Vars : Fact), New_Fact, true)
  :— !,
   make_list(Vars, LVars),
   create_variables(LVars, Internal_Vars),
   replace(Fact, LVars, Internal_Vars, New_Fact).

make_internal((Head <- Body), Head, Body) :— !.

make_internal(Fact, Fact, true) :— !.
```

```
make_list([], []) :—!.
make_list([X | Y], [X | Y]) :—!.
make_list(X, [X]).

create_variables([], []) :—!.
create_variables([Identifier | Rest_Identifiers], [Var | Rest_Vars])
  :—!,
   create_variables(Rest_Identifiers, Rest_Vars).


replace(A, _, _, A)
  :—
   (integer(A);var(A)), !.

replace((A & B), Vars, Internal_Vars, (New_A , New_B))
  :—!,
   replace(A, Vars, Internal_Vars, New_A),
   replace(B, Vars, Internal_Vars, New_B).

replace([], Vars, Internal_Vars, []).

replace([First | Rest], Vars, Internal_Vars, [New_First | New_Rest])
  :—!,
   replace(First, Vars, Internal_Vars, New_First),
   replace(Rest, Vars, Internal_Vars, New_Rest).

replace(A, Vars, Internal_Vars, New_A)
  :—
  atom(A), !,
   look_up(Vars, A, Internal_Vars, New_A).

replace(A, Vars, Internal_Vars, New_A)
  :—
  A =..[Operator | Args], !,
   replace(Args, Vars, Internal_Vars, New_Args),
   New_A =..[Operator | New_Args].



look_up([], A, _, A) :— !.

look_up([A | _], A, [New_A | _], New_A) :— !.

look_up([_ | Rest_Vars], A, [_ | Rest_Internal_Vars], New_A)
  :—!,
   look_up(Rest_Vars, A, Rest_Internal_Vars, New_A).



'@# % % '(0).

theory_gensym(M)
  :—
   retract('@# % % '(N)),
  M is N+1,
   assert('@# % % '(M)).
```

```
append([], X, X).
append([H | T], Y, [H | Z])
    :—
    append(T, Y, Z).

setOf(Templ, Gs, L) :—
    setof(Templ, Gs, L), !.
setOf(Templ, Gs, []).

xassert(Clause,DBRef) :— xasserta(Clause,DBRef).
xasserta(Clause,DBRef) :—
       builtins:get_head_information(Clause,TransformedClause,Module,ProcName,Arity),
      $compile_clause(Module,TransformedClause,code,DBRef),
     $dbref_asserta(Module,ProcName,Arity,DBRef).
```

12. Semantic Foundations.

Subsection 13.1: A Formal Deduction Calculus

To precisely specify the metaProlog system, we first specify the mathematical formal system of which the computational system is an implementation. The specifications (4.2) of Part B fix the language of the system. To complete the specification of the formal system, we must supply the rules of proof. Note that just as with the specification (4.2), the following specification of the rules of proof of mP takes place in a language (technical English) functioning as a metalanguage for mP. First we must define some auxilliary notions.

Definition 13.1. An *environment* is a finite set (table) of ordered pairs whose first element is a logical variable of mP and whose second element is a term of mP such that no two ordered pairs have the same first element (i.e., the set defines a function or mapping).

If E is an environment and X is a logical variable, we will say that E is defined on X or X is defined in E if there is some term T such that the ordered pair <X,T> belongs to E.

Definition 13.2. Environment E2 is an *extension* of environment E1 provided that E1 is a subset of E2.

Definition 13.3.

(.1) An *expression* is either a term or a literal.

(.2) If X and Y are sets, we will write X & Y for the union of X and Y. If X is a set and z is a possible element of X, we will write X & z for $X \cup \{z\}$.

(.3) An atom <U> *occurs freely* in a clause

[<V1>,...,<Vn>]:M

provided that <U> occurs in M and is distinct from each of <V1>,...,<Vn>.

(.4) If C is the clause

[<w1>,...,<wn>]:M

and D is the formula

[<v1>,...,<vn>]:N,

if none of <w1>,...,<wn>, occur freely in N, and M results from N by simultaneously replacing vi by wi for i = 1,...,n, we say that C is a *variant* of D.

(.5) We say that a clause C is *in a theory* T if there is a variant C' of C which satisfies the following:

i) T is D & S, and either C' is D or C' is in S;
ii) T is T1 & T2 and C' is either in T1 or is in T2;
iii) T is [D | L], and either C' is D or C' is in L.

Definition 13.4. Let A and B be expressions and let E1 and E2 be environments. The relation

match(A, B, E1, E2) (13.5)

is defined (at a level meta to mP) recursively as follows:

(.1) match(A, B, E1, E2) holds if A and B are identically the same constant and E2 is identical with E1.
(.2) match(A, B, E1, E2) holds if A is a logical variable which is not defined in E1 and E2 is E1 + <A,B>.
(.3) match(A, B, E1, E2) holds if A is a logical variable which is defined in E1 with <A, T> in E1 and match(T, B, E1, E2) holds.
(.4) match(A, B, E1, E2) holds if B is a logical variable which is not defined in E1 and E2 is E1 + <B,A>.
(.5) match(A, B, E1, E2) holds if B is a logical variable defined in E1 with <B, T> in E1 and match(A, T, E1, E2) holds.
(.6) match(A, B, E1, E2) holds if A and B are of the forms

<op>(C1,...,Cn) and <op>(D1,...,Dn),

respectively, and match_list([C1,...,Cn], [D1,...,Dn],E1,E2) holds.
(.7) match_list(L1, L2, E1, E2) holds if L1 and L2 are both empty and E1 is

identical with E2.

(.8) match_list(L1, L2, E1, E2) holds if the heads of L1 and L2 are H1 and H2, respectively, the tails of L1 and L2 are T1, and T2, respectively, if

match(H1, H2, E1, E3)

holds, and if

match_list(T1,, T2, E3, E2)

holds.

(.9) The only conditions under which match or match_list hold are those specified by (.1)-(.8) above.

Definition 13.6. If E and F are expressions, C1,...,Cn are logical variables, and T1,...,Tn are terms, then

subst(E, C1,...,Cn, T1,...,Tn, F) (13.7)

holds if and only if F is the expression resulting from the simultaneous substitution of T1,...,Tn for C1,...,Cn throughout E.

We will assume that the logical variables of mP can be enumerated in some fixed order. Expressions such as "the first n logical variables ..." refer to this ordering.

Definition 13.8. A *d-expression* is an expression (meta-level to mP) of the form:

d(T, G, E, P) (13.9)

where:

T is theory of mP;
G is a goal of mP;
E is an environment;
P is a list (possibly) empty of expressions of the form s(T,G,E,R), where R is either a clause or various constants.

Definition 13.10. A d-expression

d(T, <empty>, E, P) (13.11)

is said to be *terminal.*

Definition 13.12. An *m-derivation* is a finite sequence of d-expressions such that the last d-expression in the sequence is terminal and each d-expression in the sequence after the first follows from the preceeding by one of the rules of inference (13.21) - (13.yy) listed below.

Definition 13.13. If G is a primitive goal:
(.1) If G is a literal, then

selection(G, G, <empty>) (13.14)

holds.

(.2) If G is the primitive goal (H,J) where H is a literal, then

selection(G, H, J) (13.15)

holds.

(.3) If G is the primitive goal (H,J) where H is not a literal, and if

selection(H, K, L)

holds, then

selection(G, K, (L,J))

holds.

Definition 13.17. If G is a primitive goal, A <— B is a rule matrix, and

selection(G, H, K)

holds, then

transform(G, A <-- B, (B, K)) (13.18)

holds, where if K is <empty>, then (B, K) is B.

Definition 13.19. If G is a primitive goal, A is a fact matrix, and

selection(G, H, K)

holds, then

transform(G, A, K) (13.20)

holds.

Inference Rule 13.21.

$$\frac{d(T, G, E, P)}{d(T, G', E', P')}$$

provided there exists a clause

[<c1>,...,<cn>] : C

in T such that if <D1>,...,<Dn> are the first n logical variables of mP not occurring in G, E, or P (briefly, are unused), if

subst(C, <c1>,...,<cn>, <D1>,...,<Dn>, C') (13.22)

holds, if the head of C' is A', if

select(G, H, J) (13.23)

holds, then

match(H, A, E, E') (13.24)

holds, and

transform(G, C', G') (13.25)

and P' is

[s(T, G', E', [<c1>,...,<cn>] : C) | P]. (13.26)

Inference Rule 13.27.

$$\frac{d(T, demo(T', G'), E, P)}{d(T', G', E, [s(T', G', E, reflection) \mid P])}$$

Inference Rule 13.28.

$$\frac{d(T, current(U), E, P)}{d(T, current(U), E', [s(T, current(U), E', current) \mid P])}$$

where

$$match(T, U, E, E') \tag{13.29}$$

holds.

Inference Rule 13.30.

$$\frac{d(T, var(U), E, P)}{d(T, \langle empty\rangle, E, [s(T,\langle empty\rangle,E,var) \mid P])}$$

where U is a logical variable of mP and

$$dereference(U, W, E) \tag{13.31}$$

holds, where dereference is defined as below, and W is a logical variable of mP.

Definition 13.32. Let U be a logical variable of mP and let E be an environment. Then

$$dereference(U, W, E) \tag{13.33}$$

is defined as follows:

(.1) If U is not defined in E, then W is U.

(.2) If U is defined in E, say (U, V) is on E, then
  (.2.1) If V is not a logical variable of mP, W is V;
  (.2.2) If V is a logical variable, W must satisfy

dereference(V, W, E). (13.34)

Finally, we specify that the only goals that can be directly submitted to the metaProlog interpreter are those of the form

demo(T, G, E, P).

Definition 13.35. A metaProlog goal demo(T, G, E, P) is *solvable* if there exists an m-derivation whose first element is d(T, G, E, P).

14. Implementation Considerations.

Our approach to implementation has proceeded through three phases:

(1) Implementations built on top of ordinary Prolog.

(2) An interpreter written in C.

(3) A compiler derived from Warren's work [ ] on Abstract Prolog Machines (APMs).

As can be seen in Section 12 and in the Appendix to Section 11, it is possible to effectively use ordinary Prolog to create interpreters for parts of a metaProlog system. However, this creates a double layer of interpretation, resulting in far too poor performance. Moreover, the only reasonable way to carry out such implementations is to identify the variables of the language being implemented (in this case, metaProlog) with the Prolog variables. But this identification looses far too much of the metaProlog subtlty and bars us from a full implementation of the system.

Besides the desire for efficiency, the progression from one stage to the next has be driven heavily by two factors:

(1) The subtlety of the treatment of the transition from formula terms as data objects to terms as code objects and the effects of the introduction of notions of concurrency. The details of these problems run as follows:

Existing Prolog implementations trade cleverly on an ambiguous treatment of program expressions, allowing them at one an the same time to be treated as terms which can appear as arguments to predicates and simultaneously as literals (predicate calls), clause heads, and clauses themselves. Much as we have struggled to take advantage of this trick, the subtlety of the metaProlog system has eventually led us to partially abandon the device, which caused us to redesign some parts of the basic system. The fundamental difficulty lies in the treatment of variables. Standard Prolog systems identify object level and meta level variables, effectively rendering all variables of the system to be of a limited meta level kind. This causes difficulty at two closely related points: the full recursive invocation of the demonstrate predicate (the interpreter) and the proper implementation of the streamOf (setof) construct (the set-as-list of all entities satisfying a given condition). To obtain the correct handling of

environments for both of these constructs, careful distinction between the object level and meta level variables must be maintained, and this is not possible with the standard approach.

(2) Concurrency was introduced both to provide a powerful programming construct and to allow for the proper treatment of the interaction between the interpreter and certain built-in predicates for manipulating theories (or contexts). For example, the predicate addTo(T1, A, T2) holds if theory T2 is the result of adding assertion A to theory T1. The difficulty arises in the interaction between the interpreter's sequencing of predicate calls and execution of such a predicate. If at the actual run-time invocation of addTo, the assertion A has not been fully instantiated to an object level term (i.e., it still contains meta level variables), the action of the predicate is not well-defined. In general, it is not known whether the programmer intends a partially-instantiated theory to be created, whether this lack of instantiation is because some predicate call which has not yet been executed will fully instantiate A. Without concurrency and real first-class theories which can be partially instantiated, it would appear that the only choice is to treat this situation as a run-time error. But to treat this as a run-time error is far too brutal an approach for a logic-based system. As an alternative to the full introduction of partially instantiated theories, by introduction of concurrency facilties, we allow such a call to addTo to suspend, awaiting full instantiation of A. (Similar problems arise if T1 is not fully instantiated. We are also pursuing the full introduction of partially instatiated theories. However, this leads us to the notion of compiled code which is only partially instatiated, and ultimately, to a limited notion of unification of the compiled code representing formula terms.) Since it is desired that metaProlog be as close as possible in spirit to standard sequential Prolog, we are not able to use all the techniques of implementation available for Concurrent Prolog or PARALOG. In particular, we cannot make use of the read-only variable construct without far-reaching effects on both the philosophy and implementation of the system. Consequently, we were forced to make a careful analysis of the intended uses of the concurrency constructs, both as programming tools and for control of the interation of the interpreter with the indicated built-ins. The result was been a notion of *producer variable* which appears to provide the desired programming tools, maintains the basic spirit of standard Prolog, and appropriately controls the interation of the interpreter and the problematic built-ins.

For the interpreter written in C, a unifier, low-level storage allocation

routines, and a searcher (the control portion of the interpreter) were designed, and the first two fully coded and tested before we reached the conclusion that it was necessary to move on to stage (3).

The fundamental reason for moving to stage (3) -- a compiler-based system -- was efficiency. Writing an interpreter in C gave us sufficient control to solve the subtle difficulties. And while C is efficient, interpreters for Prolog simply will not produce the speeds necessary to run realistic large-size experiments.

Consequently, in the final year of the project, we embarked on the development of a compiler for metaProlog. Due to the significant problems mentioned above, there were no known compiler construction techniques which we could draw on for the entirety of the project. Instead, we have had to develop substantially new ideas and techniques as we progressed.

Since metaProlog is an (albeit significant) extension of Prolog, we began with Warren's ideas for the compilation of Prolog (cf. .....), using the copy-term approach (cf.....). No versions of such a Prolog compiler were available to us in source form, so our first task was to construct a solid version of such a Prolog compiler to use as the basis of our further work. We first set out to proceed by a classic bootstrap, writing the compiler in Prolog, and booting it on itself using C-Prolog on a VAX 780. We joined forces with the group led by Ross Overbeek and Rusty Lusk at Argonne National Labs, who had written an implementation of a copy-stack Warren abstract machine (WAM) in C as part of their exploration of concurrency in Prolog. We succeeded in writing the compiler, but the limitations of our C-Prolog and our then current UNIX system on the VAX made this incredibly awkward, forcing us to segment the compiler sources in many small source files. Moreover, the performance of the Argonne WAM left much to be desired. Thus, we rewrote the entire system in C, providing the performance we desired (Bowen, et al. [1985]). Of special concern was the ability to dynamically handle compiled code as required by the dynamic Prolog database predicates "assert and retract". No previous system had been able to deal with these, and forced programmers to declare predicates which would be acted on by assert or retract , in which case the system kept these predicates interpreted, causing considerable slowdown. The case of assert was not difficult to deal with, since we simply included our C-coded compiler as a callable built-in predicate -- its performance made this eminently sensible. However, the case of retract is very different. One is given a term - i.e., Prolog structure - and must located the head of a compiled clause which matches that structure. The group developed a new

technique of decompilation to deal with this problem (Buettner [1985]). It is unlike traditional decompilation techniques which pick apart the compiled code to try to infer the source code which produced it. Instead, it exploits the fact that compiled Prolog still represents the pattern-matching the unification carries out during Prolog execution. Simply put, by running the compiled clause of the code in an odd mode, we force it to build a copy of the source from which it was compiled. (NO special compilation mode is used.) Then the head of the decompiled clause is matched against the term passed to retract. Along the way, we discovered a number of compiler optimizations which were previously unknown (Turk [1985]).

We have begun developing serious designs which will guide us in reshaping the present Prolog compiler (dubbed Columbus Prolog) to become a compiler for metaProlog. As we see it now, most of the effects will take place in the underlying abstract Prolog machine -- most of the process of compilation of metaProlog clauses to instructions for the new machine will be almost identical to that for Prolog. We must not only support theories as first-class data objects, but must move compiled code from a separate code space to exist on the abstract Prolog machine heap. All of this will have profound impact on the garbage collection process. Our current thinking, which is well-developed, is partially reflected in the simulators shown in Section 12. We plan to continue this work under the RADC Artificial Intelligence Consortium grant.

REFERENCES

Bowen, K.A. *Meta-level programming and knowledge representation*, **New Generation Computing**, v.3 (1985), pp.359-383.

Bowen,K.A., Buettner,K.A., Cicekli,I., & Turk,A.K., *The design and implementation of a high-speed incremental portable Prolog compiler*, **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

Bowen, K.A., and Kowalski,R.A., *Amalgamating language and metalanguage in logic programming*, in **Logic Programming**, Clark and Tarnlund, eds, Academic Press, 1982.

Bowen, K.A. & Weinberg,T., *metaProlog: A metalevel extension of Prolog*,**Proc. 1985 Symp. on Logic Programming**, Boston, 1985.

Buettner,K.A., *Fast decompilation of compiled Prolog clauses*, **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

Duda, R., Hart,P., Barrett,P., Gaschnig,J., Konolige,K.,Reboh,R., and Slocum,J., **Development of the Prospector system for mineral exploration**, Final Report, SRI International, Menlo Park, CA, 1978.

Fain, J., Hayes-Roth, F., Sowizral, H. & Waterman, D., *Programming in ROSIE: An Introduction by Means of Examples*, Report N-1646-ARPA, Rand Corp., 1982.

Hayes-Roth, F., Lenat, D., & Waterman, D., **Building Expert Systems**, Addison-Wesley, 1984.

Kowalski, R.A. *Logic as a database language*, preprint, Imperial College, London, July, 1981.

Nicholas, J.M., and Gallaire, H., *Database: theory vs. interpretation*, in **Logic and Databases**, ed. Gallaire and Minker, Plenum Press, New York, 1978.

Reiter, R., *Towards a logical reconstruction of relational databasse theory*, preprint, Dept. of Computer Science, Univ. of British Columbia, 1981.

Shoenfield, J.R., **Mathematical Logic**, Addison-Wesley, Reading, Ma.., 1967.

Turk, A.K., *Compiler optimizations for the WAM*, **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986.

Warren, D.H.D., *An abstract Prolog instruction set,* SRI Technical Report, 1983.

Warren, D., Pereira, L., and Pereira. F., *Prolog--The language and its implementation compared the LISP*, **SIGPLAN Notices** 12., no. 8, 1977, 109-115 (also **SIGART Newsletter**, no.64).

Weyhrauch,R. *Prolegomena to a theory of mechanized formal reasoning*, **Artifical Intelligence**, 13, 133-170.

END

12-87

DTIC